Entwicklung eines visuellen Editors für Anwendungsbedingungen und amalgamierte Regeln zur Flexibilisierung von EMF-Modelltransformationen

Diplomarbeit

Angeline Warning

Matrikel-Nr. 227696

Berlin, 23. Dezember 2010

Technische Universität Berlin

Fakultät IV – Elektrotechnik und Informatik

Institut für Softwaretechnik und Theoretische Informatik

Fachgebiet Theoretische Informatik / Formale Spezifikation (TFS)

Einsteinufer 17

10587 Berlin

Deutschland

Betreuer:

Prof. Dr. Hartmut Ehrig

Dr. Claudia Ermel

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 23. Dezember 2010

Angeline Warning

Vorwort

Diese Diplomarbeit entstand an der Technischen Universität Berlin am Fachbereich für Theoretische Informatik / Formale Spezifikation (TFS).

Mein Dank gilt allen, die mich mit Rat und wertvollen Diskussionsbeiträgen bei der Arbeit unterstützt haben: Prof. Dr. Hartmut Ehrig, Dr. Claudia Ermel für die Betreuung meiner schriftlichen Arbeit und Dipl. Ing. Enrico Biermann für die Betreuung meiner praktischen Arbeit.

Besonderer Dank geht außerdem an meinen Mann Sacha Warning, der mich mit viel Verständnis unterstützt und mit viel Geduld meine Arbeit gelesen und korrigiert hat. Ohne ihn wäre diese Arbeit nicht möglich gewesen.

Bedanken möchte ich mich auch bei meinen Kindern Sophie und Leonard und bei meiner Schwester Angela, die viel Verständnis für diese umfangreiche Arbeit aufgebracht haben.

Vielen Dank.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Wintersemester 2009/2010 wurde das Projekt Visuelle-Sprachen am Fachbereich Theoretische Informatik / Formale Spezifikation (TFS) unter der Anleitung von Dr. Claudia Ermel und Enrico Biermann an der Technischen Universität Berlin durchgeführt. Ziel des Projekts war es, einen Editor für Graphtransformationen auf Basis der visuellen Sprache Henshin zu entwickeln. Das Ergebnis war der Henshin-Editor, der einen baumbasierten und graphischen Editor beinhaltet, mit dem Graphen und Transformationsregeln visuell definiert werden können.

Die Ausbaustufe, die im Rahmen des Projektes erreicht wurde, unterstützte noch nicht den gesamten Umfang der Sprache Henshin, was den Nutzen des Editors erheblich einschränkt.

Diese Arbeit umfasst zwei Erweiterungen, die zur Flexibilisierung von EMF-Modelltransformationen beitragen sollen. Zum einen soll der Editor dahingehend erweitert werden, dass auch komplexe und verschachtelte Anwendungsbedingungen definiert werden können. Zum anderen soll es möglich sein, amalgamierte Regeln zu definieren.

Komplexe Anwendungsbedingungen sind Anwendungsbedingungen, die mit den logischen Operatoren *NOT*, *AND* und *OR* verknüpft werden. Die Verschachtelung von Anwendungsbedingungen kann dadurch erreicht werden, dass eine Anwendungsbedingung über einer anderen definiert wird.

Eine amalgamierte Regel ist eine normale Transformationsregel, die den Effekt der Anwendung einer Amalgamation-Unit beschreibt. Sie wird aus einer Amalgamation-Unit und einem Graphen konstruiert. Amalgamierte Regeln ermöglichen eine variable Anzahl von Transformationen, ohne dass diese vorher festgelegt werden muss.

Zu Beginn der Arbeit werden im Kapitel 2 einige Grundkenntnisse zu Graphen sowie die Sprache Henshin erläutert. Kapitel 3 enthält Informationen über den Henshin-Editor selbst, seine Entstehung, seinen Aufbau und über den Ist-Stand als Basis für diese Arbeit.

Kapitel 4 behandelt die Themen Anwendungsbedingungen und amalgamierte Regeln und erläutert die Probleme, die ohne die im Rahmen dieser Arbeit entwickelten Erweiterungen auftreten können. Hier werden auch die konkreten Anforderungen an den erweiterten Henshin-Editor aufgelistet.

Die gewählten Lösungsansätze bezüglich der gestellten Anforderungen werden im Kapitel 5 vorgestellt und möglichen Alternativen gegenübergestellt. Kapitel 6 stellt die Technologien vor, die für die Entwicklung des Henshin-Editors verwendet wurden. Hier wird der Implementierungsteil näher betrachtet. Kapitel 7 zeigt ein umfangreiches Statechart als Anwendungsbeispiel für den erweiterten Henshin-Editor.

Am Ende werden im Kapitel 8 einige verwandte Arbeiten erläutert. Den Abschluß bildet Kapitel 9, das die Ergebnisse dieser Arbeit zusammenfasst und einen Ausblick auf geplante Erweiterungen gibt.

2 Grundlagen

In diesem Kapitel werden zuerst einige Begriffe erläutert, die im Zusammenhang mit Graphtransformationen stehen. Sie dienen dem besseren Verständnis des 3. Kapitels. Anschließend wird die Sprache Henshin erklärt, für die der Editor entwickelt wurde, sowie das Basismodell von Henshin und das Henshinlayout. Ausführliche Informationen über den Editor sind im Kapitel 3 zu finden. Das Kapitel endet mit einer Erklärung über gültige Graphen im Bezug auf EMF-Modelle.

2.1 Graph

Im Allgemeinen ist ein (ungerichteter) Graph $G = (V^G, E^G)$ definiert durch:

- eine endliche Menge von Knoten V *(engl. vertex)*
- eine endliche Menge von Kanten E *(engl. edge)*, die die Knoten miteinander verbinden $E^G \quad V^G \cdot V^G$

Ein ungerichteter Graph hat symmetrisch gerichtete Kanten, d.h. eine Kante vom Ursprungsknoten zum Zielknoten ist gleich zu bewerten wie eine Kante vom Zielknoten zum Ursprungsknoten.

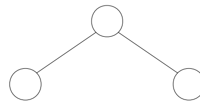


Abbildung 1: (Ungerichteter) Graph

2.1.1 Gerichteter Graph

Ein gerichteter Graph $G = (V^G, E^G, src^G, tar^G)$ ist ein Graph mit gerichteten Kanten. Er besteht aus:

- einer endlichen Menge von Knoten V^G *(engl. vertex)*
- einer endlichen Menge von Kanten E^G *(engl. edge)*, die die Knoten miteinander verbinden $E^G \quad V^G \cdot V^G$
- einer Abbildung $src^G : E^G \rightarrow V^G$, die einer Kante einen Ursprungsknoten zuweist
- einer Abbildung $tar^G : E^G \rightarrow V^G$, die einer Kante einen Zielknoten zuweist

In diesem Fall haben eine Kante vom Ursprungsknoten zum Zielknoten und eine vom Zielknoten zum Ursprungsknoten unterschiedliche Bedeutungen.

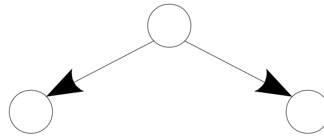


Abbildung 2 : Gerichteter Graph

2.1.2 Graphmorphismus

Ein Graphmorphismus besteht aus Abbildungen, die Start- und Zielknoten bewahren. Er definiert Bedingungen, wann ein Abbild (Match) eines Graphen in einem anderen Graphen gefunden werden kann.

Formal definiert: Ein Graphmorphismus $f : G \rightarrow H$ zwischen den Graphen G und H mit

$$G = (V^G, E^G, src^G, tar^G) \quad \text{und} \quad H = (V^H, E^H, src^H, tar^H)$$

besteht aus Abbildungen

$$f_V : V^G \rightarrow V^H \quad \text{und} \quad f_E : E^G \rightarrow E^H,$$

d.h. $f_V(src^G(e)) = src^H(f_E(e))$ und $f_V(tar^G(e)) = tar^H(f_E(e))$, wobei $e \quad E^G \quad E^H$.

Beispiel: Abbildung 3 zeigt einen Graphmorphismus von G nach H, da alle Knoten und Kanten von G in H abgebildet sind. Dagegen existiert von G nach I kein Graphmorphismus, da die Kante b von G in I nicht abgebildet werden kann.

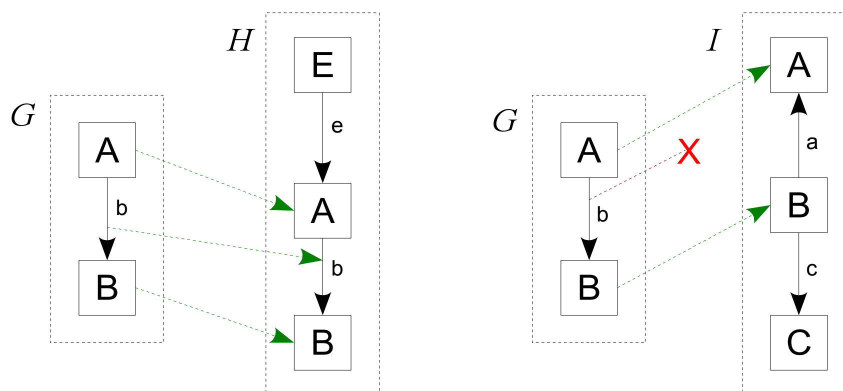


Abbildung 3: Graphmorphismus von G nach H

2.1.3 Typisierter Graph

Ein typisierter Graph $TG = (T^V, T^E, src^T, tar^T)$ ist ein Graph G zusammen mit einem Graph-morphismus $t^G = G \rightarrow TG = (t_V : V^G \rightarrow T^V, t_E : E^G \rightarrow T^E)$ mit:

- $src^T, tar^T : T^E \rightarrow T^V$ (Abbildungen von Ursprungs- und Zielknoten)
- T^V : eine endliche Menge von Knotentypen
- T^E : eine endliche Menge von Kantentypen
- einer Abbildung $t_V : V^G \rightarrow T^V$, die einem Knoten einen Typ aus T^V zuweist
- einer Abbildung $t_E : E^G \rightarrow T^E$, die einem Knoten einen Typ aus T^V zuweist

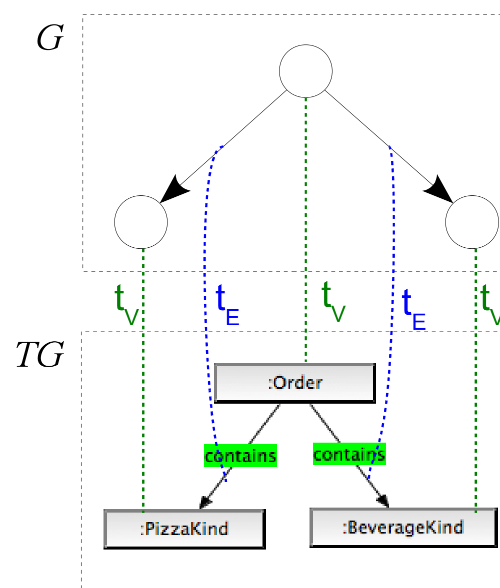


Abbildung 4: Typisierter Graph und Graphmorphismus

2.1.4 Attributierter Graph

Ein attributierter Graph ist ein Graph, dessen Knoten und Kanten um Attribute erweitert werden können.

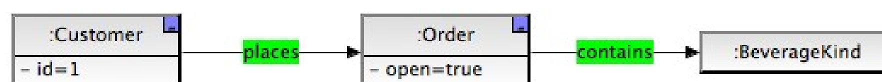


Abbildung 5 : Attributierter getypter gerichteter Graph

2.1.5 Graphen mit Mehrfachkanten

Graphen mit Mehrfachkanten sind Graphen, die Knotenpaare beinhalten, die durch mehr als eine Kante miteinander verbunden sind.

Für diese Arbeit sind Graphen relevant, die attributiert, typisiert und gerichtet sind und zudem Mehrfachkanten besitzen können.

2.1.6 Teilgraph

Ein Graph G heißt Teilgraph von H, $G \subseteq H$, wenn alle Knoten bzw. Kanten, die in G existieren, in H enthalten sind, also $V^G \subseteq V^H$ und $E^G \subseteq E^H$.

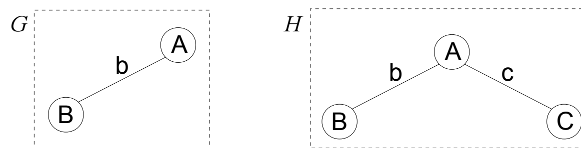


Abbildung 6: G ist Teilgraph von H ($G \subseteq H$)

Bei gerichteten Graphen gilt zusätzlich:

- G und H sind vom gleichen Typ $(t^G = t^H)$
- $\forall e \in E^G$: $src^G(e) = src^H(e)$ und $tar^G(e) = tar^H(e)$

2.1.7 Vereinigung von Graphen

Ein Graph A ist eine Vereinigung der Graphen G und H, $A = G \cup H$, wenn G und H jeweils Teilgraphen von A sind und es gilt: $V^A = V^G \cup V^H$ und $E^A = E^G \cup E^H$.

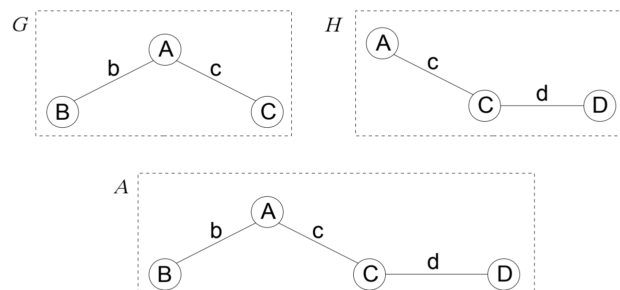


Abbildung 7: Vereinigung von G und H ($A = G \cup H$)

2.1.8 Schnitt von Graphen

Ein Graph B ist ein Schnitt der Graphen G und H, $B = G \quad H$, wenn B sowohl ein Teilgraph von G als auch von H ist und es gilt: $V^B = V^G \quad V^H$ und $E^B = E^G \quad E^H$.

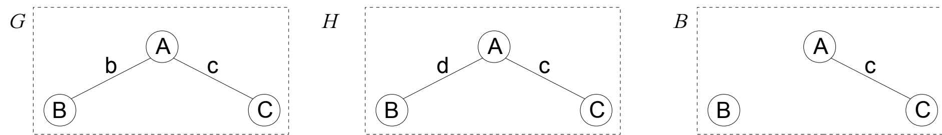


Abbildung 8: Schnitt von G und H ($B = G \quad H$)

2.2 Graphtransformation

Als Graphtransformation bezeicnet man einen Vorgang, bei dem ein Graph mit Hilfe einer Regel verändert wird.

2.2.1 Graphtransformationsregel

Eine Graphtransformationsregel ist eine Regel, die ausgeführt wird, um einen Graphen zu einem anderen Graphen zu transformieren. Sei G der Arbeitsgraph. Eine Regel hat einen Namen und ist definiert durch:

- einen Graph LHS (linke Regelseite)

  LHS ist eine Vorbedingung, die erfüllt sein muss, bevor eine Graphtransformation durchgeführt wird. Diese Bedingung ist erfüllt, wenn ein Graphmorphismus von LHS in G gefunden werden kann ($f : LHS \rightarrow G$).

- beliebig viele negative Anwendungsbedingungen (NACs)

  Die negative Anwendungsbedingungen sorgen dafür, die Regelanwendung zu verbieten, wenn ein injektiver Graphmorphismus von NAC in G gefunden werden kann. Mehrere NACs werden in Henshin durch ein logisches AND verknüpft. Eine detailierte Erläuterung über Anwendungsbedingungen ist im Abschnitt 4.1 (S. 23) zu finden.

  Ein Graphmorphismus $f : LHS \rightarrow G$ wird Match genannt, wenn er alle NACs erfüllt.

- einen Graph RHS (rechte Regelseite)

In RHS wird die Nachbedingung der Transformation (bzw. der Regelanwendung) definiert.

- ein Mapping zwischen LHS- und RHS-Knoten bzw. zwischen LHS- und NAC-Knoten

  Ein Mapping zwischen zwei Knoten gleicher Typen stellt denselben Knoten im Arbeitsgraphen dar.

Beispiel: Anwendung einer Graphtransformationsregel auf G

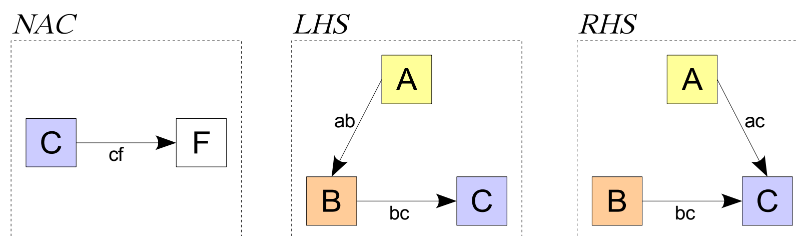


Abbildung 9: Beispiel einer Graphtransformationsregel

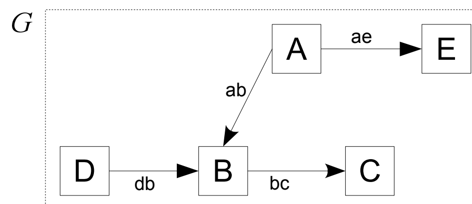


Abbildung 10: Arbeitsgraph G, auf den die Regel angewendet wird

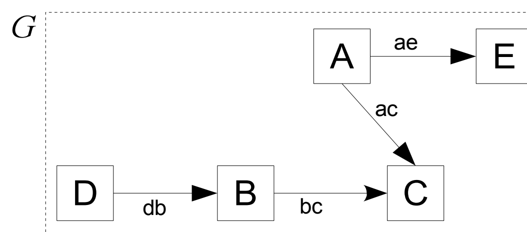


Abbildung 11: Graph G nach der Regelanwendung

Folgende Schritte werden bei der Regelausführung durchgeführt:

1. Es wird zunächst überprüft, ob ein Match m von LHS in G existiert. Nur wenn dies der Fall ist, werden die weiteren Arbeitsschritte durchgeführt (siehe Abbildung 12).

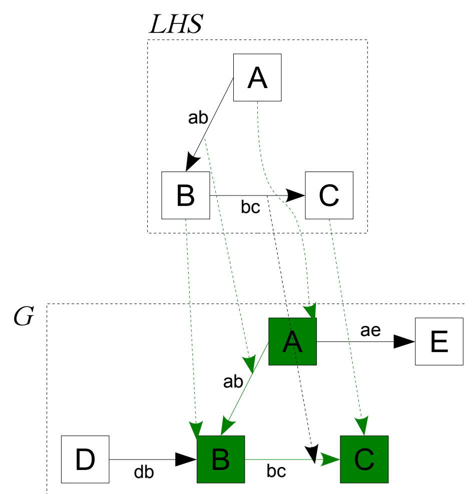


Abbildung 12: Graphmorphismus $m : LHS \rightarrow G$

2. Sind NACs in der Transformationsregel definiert, wird überprüft, ob ein Graphmorphismus von einem der $NACs$ im Arbeitsgraphen existiert. Ist dies der Fall, wird die Regelausführung verboten. Die Überprüfung auf NAC-Morphismen wird nur durchgeführt, wenn der Schnitt $LHS \quad NAC$ definiert ist. Gibt es für keine NAC Graphmorphismen $n : NAC \rightarrow G$, dann ist $m : LHS \rightarrow G$ ein Match. (siehe Abbildung 13)

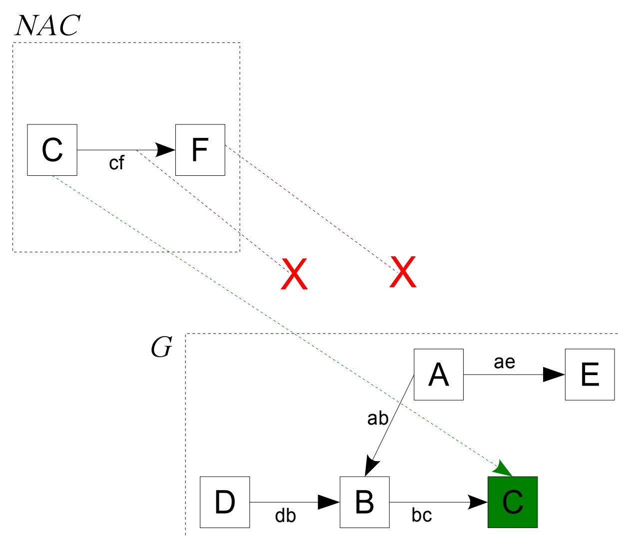


Abbildung 13: Kein NAC-Morphismus $n : NAC \rightarrow G$ vorhanden

3. Sei $S := LHS \quad RHS$. Das Matchteil $m(LHS)$ in G wird durch S ersetzt, d.h. alle Graphelementen in $m(LHS)$, die in S nicht vorkommen, werden gelöscht.

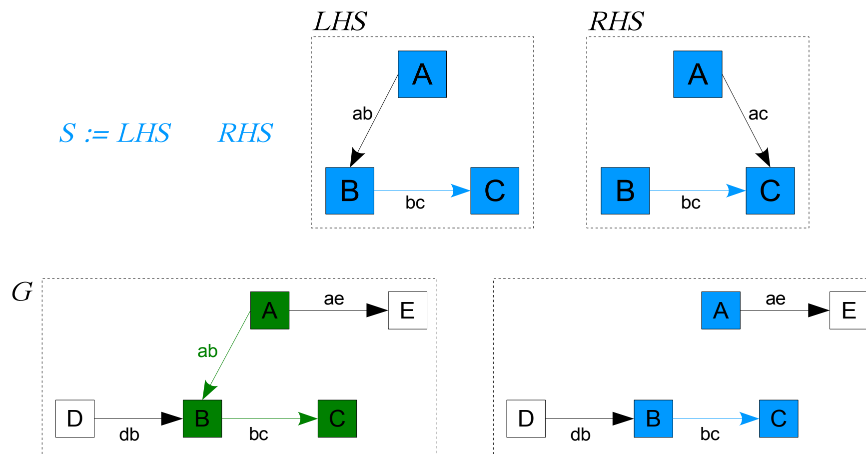


Abbildung 14: Ersetzen von M durch S

4. Vereinigung von G und RHS.

2.3 Henshin

Das Wort „Henshin" stammt aus Japan und heißt „Transformation". Henshin ist eine Sprache, die die visuelle Modellierung und die Ausführung von regelbasierten EMF-Modell-Transformationen unterstützt.

Für Henshin existiert ein gleichnamiges Eclipse-Projekt *[ECL10]*, das die Sprache in Eclipse als Plug-in integriert. Dies ermöglicht eine formale Verifikation von Modelltransformationen.

Henshin verändert Instanzgraphen „in-place". Das bedeutet, dass bei einer Transformation kein neuer Graph erzeugt wird, sondern sie direkt in dem Instanzgraphen durchgeführt wird.

2.3.1 Henshin Basismodell

Henshin hat zwei Kernklassen: Graphen und Regeln (Transformationeinheiten). Wie in den Abschnitten 2.1 und 2.2 beschrieben wurde, setzen sich Graphen und Regeln aus mehreren Bestandteilen zusammen. Die weiter oben beschriebenen Eigenschaften von Graphen und Regeln finden sich in dem Henshin Basismodell in Form eines EMF-Modells wieder.

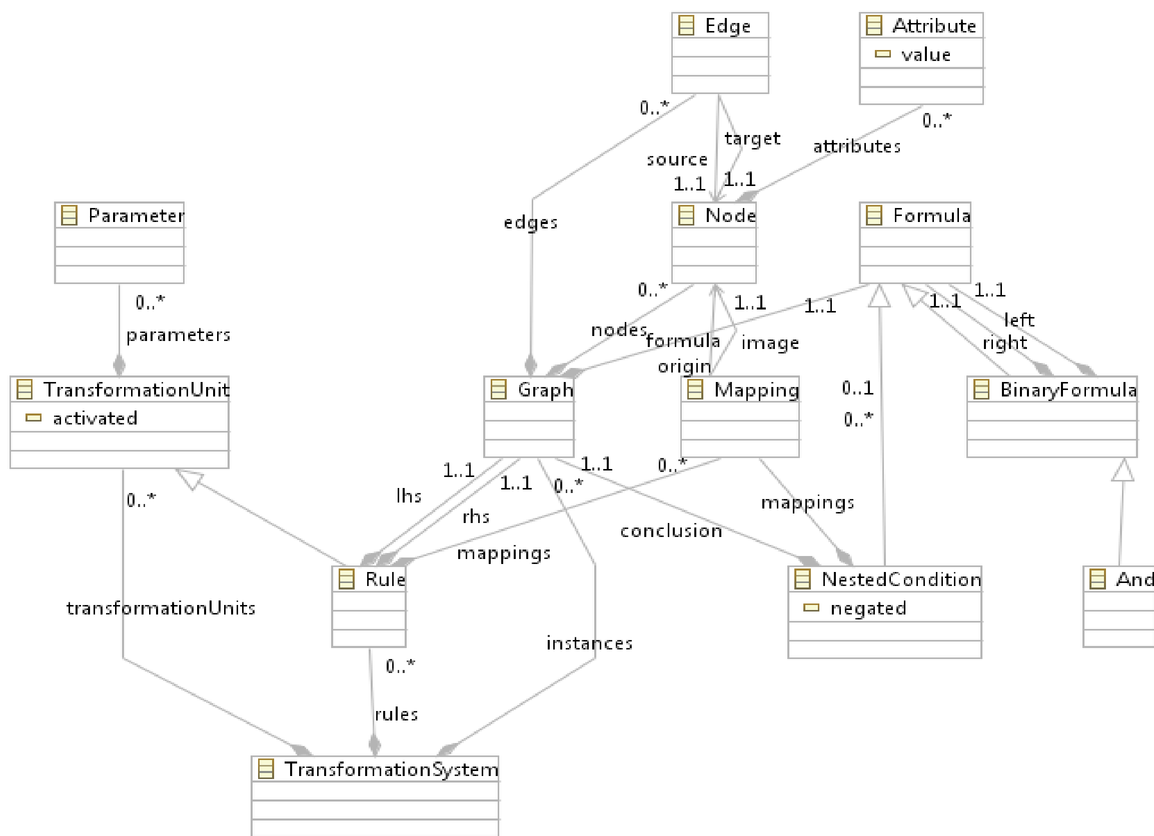


Abbildung 15: Henshin Basismodell

Im Folgenden wird das Henshin-Modell genauer erklärt:

- Ein Transformationssystem *(TransformationSystem)* kann aus mehreren Instanzgraphen, Regeln und Transformationseinheiten bestehen.
- Ein *Graph* kann beliebig viele Knoten *(Node)* und Kanten *(Edge)* haben. Optional kann ein Graph eine Formel *(Formula)* haben.
- Ein Knoten kann über mehrere Attribute *(Attribute)* verfügen.
- Jede Kante hat genau einen Quellknoten und genau einen Zielknoten.
- Eine Formel ist entweder eine Anwendungsbedingung *(NestedCondition)* oder eine binäre Formel *(BinaryFormula)*. Eine binäre Formel ist ein *And* und hat genau eine linke und eine rechte Formel.
- Eine Anwendungsbedingung kann einen Graphen beinhalten.

- Eine Regel *(Rule)* hat genau einen *LHS*-Graphen und einen *RHS*-Graphen. Sie kann auch aus mehreren Mappings bestehen, die die Schnitte *LHS* *RHS* und *LHS* *NAC* definieren. Durch die *TransformationUnit* kann eine Regel mehrere *Parameter* haben.
- Ein *Mapping* hat genau einen Ursprungsknoten und einen Abbildungsknoten.

Es ist zu beachten, dass *ApplicationCondition* im Henshin-Modell durch *NestedCondition* repräsentiert ist. Die Umbennung im Modell kann zur Zeit noch nicht durchgeführt werden, da dadurch existierende Henshindateien im Henshin-Editor nicht mehr geöffnet werden könnten.

2.3.2 Henshinlayout Basismodell

Das *LayoutSystem* kann mehrere Knotenlayouts haben. Das Knotenlayout *(NodeLayout)* dient zum Anzeigen eines Knotens im Henshin-Editor. Die Attribute *x*, *y* geben die Position des dargestellten Knotens an. Mit *hide* kann bestimmt werden, ob die Knotenattribute mit angezeigt oder versteckt werden sollen.

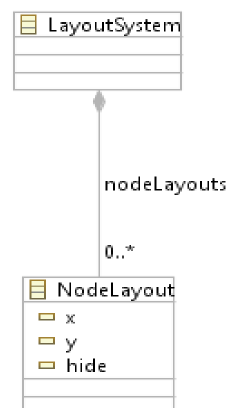


Abbildung 16: Henshin-Layout-System als EMF-Modell

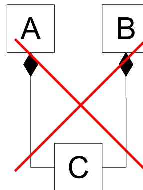
2.4 EMF-Kompatibilität

Jeder Graph mit den oben beschriebenen Eigenschaften (siehe 2.1, S. 3) ist ein gültiger Graph. Jedoch kann nicht aus jedem gültigen Graphen auch ein gültiges EMF-Instanzmodell abgeleitet werden.

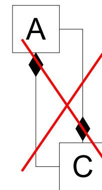
2.4.1 Konsistente EMF-Modelle

In einem konsistenten EMF-Modell hat die Containment-Hierarchie eine Baumstruktur, deren Wurzel ein Container-Knoten ist. Die Eigenschaften konsistenter EMF-Modelle sind:

- Jedes Objekt, mit Ausnahme der Wurzel, hat höchstens einen Container. Die Wurzel hat keinen Container.



- Zyklisches Containment ist unzulässig



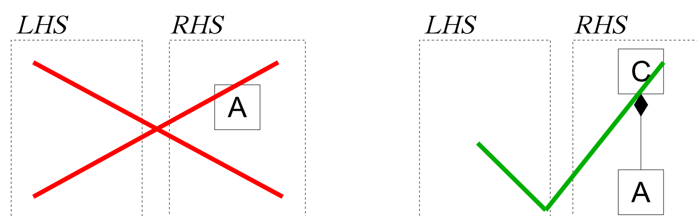
- Optional sind alle Objekte transitiv in einem Wurzelknoten enthalten. Dies bedeutet, dass jeder Instanzgraph genau eine Wurzel hat.

Im Henshin-Editor zeichnen sich die gültigen Instanzen eines EMF-Modells durch alle drei Eigenschaften aus.

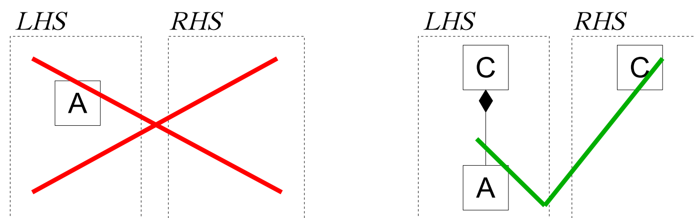
2.4.2 Konsistenz von EMF-Transformationsregeln

Bei der Graphtransformation werden Knoten oder Kanten hinzugefügt oder gelöscht. Das kann dazu führen, dass ein konsistentes EMF-Instanzmodell nach der Graphtransformation inkonsistent ist. Um dies zu verhindern, müssen die EMF-Transformationsregeln folgende Konsistenzbedingungen erfüllen *[BET08]*:

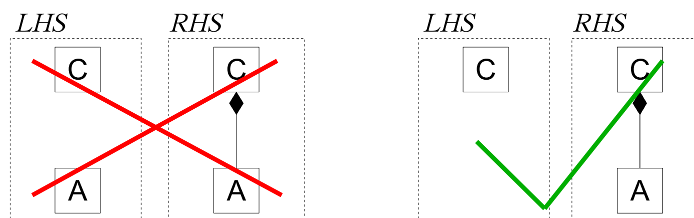
- Ein Knoten wird nur zusammen mit der zugehörigen Containment-Kante erstellt.



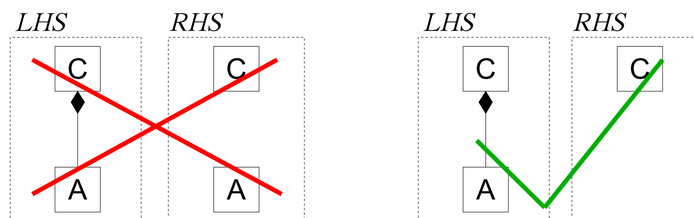
- Ein Knoten wird nur zusammen mit der zugehörigen Containment-Kante gelöscht.



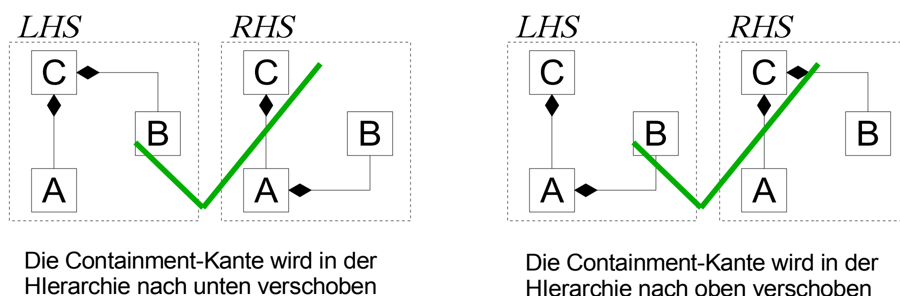
- Eine Containment-Kante wird nur zusammen mit dem enthaltenen Knoten erstellt.



- Eine Containment-Kante wird nur zusammen mit dem enthaltenen Knoten gelöscht.



- Ein Knoten darf seinen Container nur wechseln, wenn sowohl der alte als auch der neue Container transitiv im selben Container enthalten sind.



Die Containment-Kante wird in der HIerarchie nach unten verschoben

Die Containment-Kante wird in der HIerarchie nach oben verschoben

3 Henshin-Editor

Dieses Kapitel beschreibt den Henshin-Editor, der als Grundlage dieser Arbeit dient. Es beginnt mit einer Erklärung, was der Henshin-Editor ist und wie er entstanden ist. Danach werden die Anforderungen erläutert, die an den Editor gestellt werden. Anschließend wird der Ist-Stand des Editors und seine Funktionalitäten vorgestellt, bevor diese Arbeit geschrieben wurde. Das Kapitel endet mit einem Anwendungsbeispiel auf Grundlage des Ist-Standes des Henshin-Editors.

3.1 Definition und Entstehung

Der Henshin-Editor ist ein visueller Editor für eine visuelle Modellierungssprache. Er integeriert EMF-Diagramme und Graphtransformationen. Er wurde als ein Eclipse-Plug-In im Rahmen des Visuelle-Sprachen-Projekts im Wintersemester 2009/2010 entwickelt, geleitet von Dr. Claudia Ermel.

Das grundsätzliche Ziel des Henshin-Editors ist es, alle Instanzen der im Henshin-Editor importierten EMF-Modelle visuell darzustellen, zu erzeugen und zu bearbeiten. Hierbei sollen die Möglichkeiten genutzt werden, die in graphischen Benutzeroberflächen zur Verfügung stehen (z.B. „Drag and Drop“).

3.2 Henshin-Editor-Oberfläche

Die Oberfläche des Henshin-Editors setzt sich aus den folgenden Ansichten zusammen, die alle Bestandteil der Henshin-Perspektive sind:

- Baumansicht (Tree View)
- Graphansicht (Graph View)
- Regelansicht (Rule View)

3.2.1 Baumansicht (Tree View)

In der Baumansicht werden die Instanzen eines Transformationssystems als Baum dargestellt. Die Wurzel des Baumes ist der Knoten *transformation system*, der für das Transformationssystem steht. Darunter befindet sich immer ein Verzeichnis *Imported EPackage(s)*, das die importierten Modelle beinhaltet. Optional können ein oder mehrere Graphen und Regeln unter dem Transformationssystem stehen.

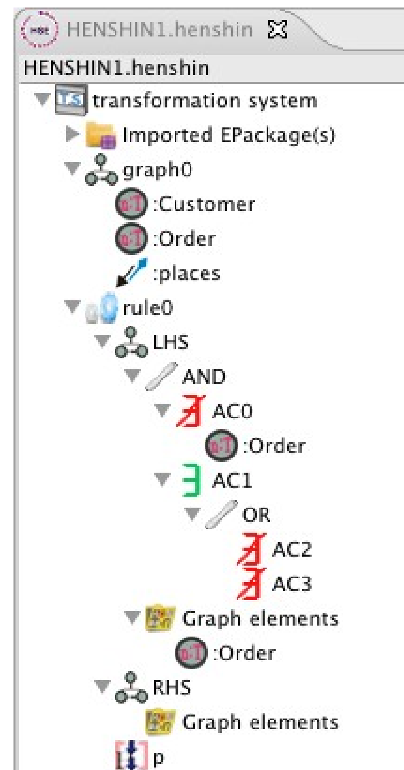


Abbildung 17: Tree View

Ein Graph kann über einen oder mehrere Einträge verfügen, die Knoten und Kanten im Graphen repräsentieren. Ein Knoten kann einen oder mehrere Einträge haben, die Attribute darstellen.

Für jede Regel existiert im Baum ein *LHS*-Eintrag, ein *RHS*- Eintrag und optional ein oder mehrere Parametereinträge. Die *LHS*- und *RHS*-Einträge haben jeweils ein Unterverzeichnis *Graph elements*, das deren Knoten und Kanten repräsentiert.

Die Parametereinträge stellen die zulässigen Werte dar, die bei der Attributzuweisung im *LHS*- und *RHS*-Graphen verwendet werden können.

Der *LHS*-Eintrag kann einen Formeleintrag beinhalten, der die Anwendungsbedingung einer Regel darstellt. Zur Formel gehören die *Application Condition* und die booleschen Ausdrücke* *NOT*, *AND* und *OR*. Es befindet sich immer ein Kindeintrag unter *NOT* und zwei Kindeinträge unter *AND* und *OR*. Die Kindeinträge eines booleschen Ausdrucks sind wiederum Formeleintrag.

Der Eintrag, der *Application Condition* repräsentiert, ist wie ein Grapheintrag zu behandeln. Er kann über einen oder mehrere Einträge verfügen, die Knoten und Kanten im Graphen darstellen. Zusätzlich kann in diesem Eintrag ein Formeleintrag eingefügt werden.

Das Editieren der einzelnen Elementen erfolgt am einfachsten über die Kontextmenüs der Baumeintrag. Sie werden durch einen rechten Mausklick auf den jeweiligen Eintrag geöffnet und enthalten die zum Eintrag passenden Menüpunkte.

Die am häufigsten benötigten Funktionen lassen sich außerdem über Schaltflächen in der Werkzeugleiste erreichen. Hierzu gehören Funktionen zum Bearbeiten von Elementen, wie z.B. Delete, Undo und Redo.

3.2.2 Graphansicht (Graph View)

In der Graphansicht wird ein Graph visuell dargestellt. Knoten und Attribute können außerdem visuell bearbeitet werden. Kanten können nur erzeugt und gelöscht werden.

* *Die booleschen Ausdrücke sind ein Teil der Erweiterung des Henshin-Editors, um die es in dieser Arbeit auch geht.*

Auf der rechten Seite befindet sich eine aufklappbare Palette mit Schaltflächen zum Bearbeiten von Graphelementen. Auf der obere Seite ist eine Leiste mit Knöpfen mit wichtigen Funktionen, wie

- Prüfen des angezeigten Graphen auf EMF-Kompatibilität (siehe 2.4, S. 12)
- Ausführen einer Regel auf dem angezeigten Graphen

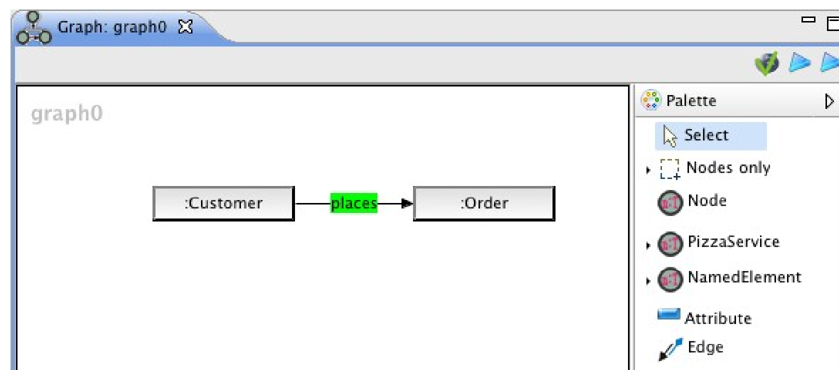


Abbildung 18: Graph View

3.2.3 Regelansicht (Rule View)

In der Regelansicht lässt sich eine Graphtransformationsregel definieren. Diese Ansicht stellt den *LHS*-Graphen, den *RHS*-Graphen und die *NAC*s dar. Im Henshin-Editor wird während des Projekts immer nur ein *NAC* neben dem *LHS* angezeigt. Zum Blättern nach links und nach rechts dienen die Pfeil-nach-links und die Pfeil-nach-rechts Taste.

Auf der rechten Seite befindet sich eine aufklappbare Palette mit Schaltflächen zum Bearbeiten von Graphelementen in der angezeigten Regel. Auf der obere Seite ist eine Leiste mit Knöpfen mit wichtigen Funktionen, wie

- Prüfen der angezeigten Regel auf EMF-Kompatibilität (siehe 2.4, S. 12)
- Ausführen der angezeigten Regel
- Kopieren eines *LHS*-Graphen zum *RHS*-Graphen und eines *LHS*-Graphen zum *NAC*-Graphen

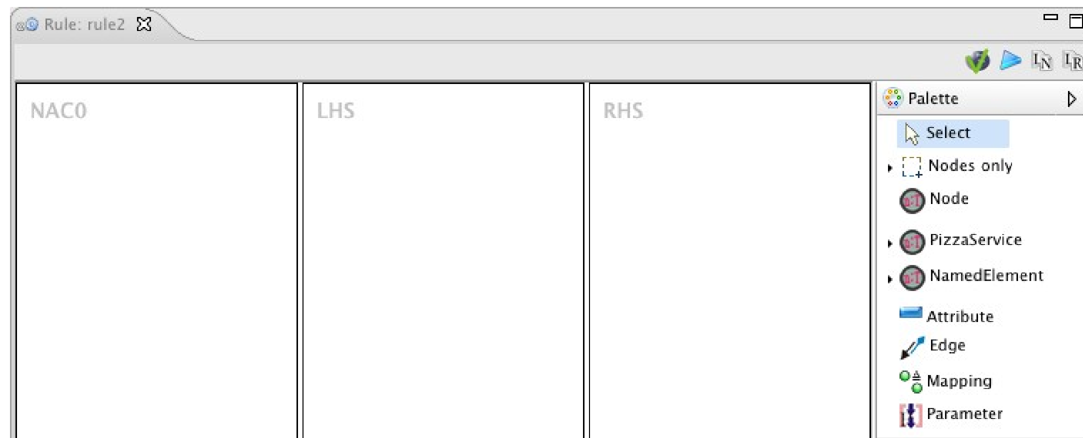


Abbildung 19: Rule View

3.2.4 Henshin Perspective

Die Henshinperspektive wird nach dem Anlegen einer Henshindatei oder nach einem Doppelklick auf eine Henshindatei automatisch geöffnet. In dieser Perspektive werden initial die vorkonfigurierten Anordnungen von Views, Menüs und Symbolleisten gezeigt. Die Anordnung der Ansichtsfenster können durch „Drag and Drop“ beliebig geändert werden. Beim Schließen des Editors wird die Anordnung der Fenster automatisch gespeichert, so dass der Nutzer beim Wiederöffnen der Henshindatei dieselbe Anordnung wieder vorfindet.

Zur Henshin-Perspektive gehören auch eine Reihe von in Eclipse bekannten Standardansichten, wie z.B.:

- Properties View

  Im Properties-View werden die Eigenschaften von einem aktuell ausgewählten Objekt angezeigt. Sofern es erlaubt ist, kann der Wert (*Value*) einer Eigenschaft (*Property*) an dieser Stelle geändert werden.

- Navigator View

  Im Navigator-View werden alle Dateien angezeigt, die im Projektverzeichnis enthalten sind.

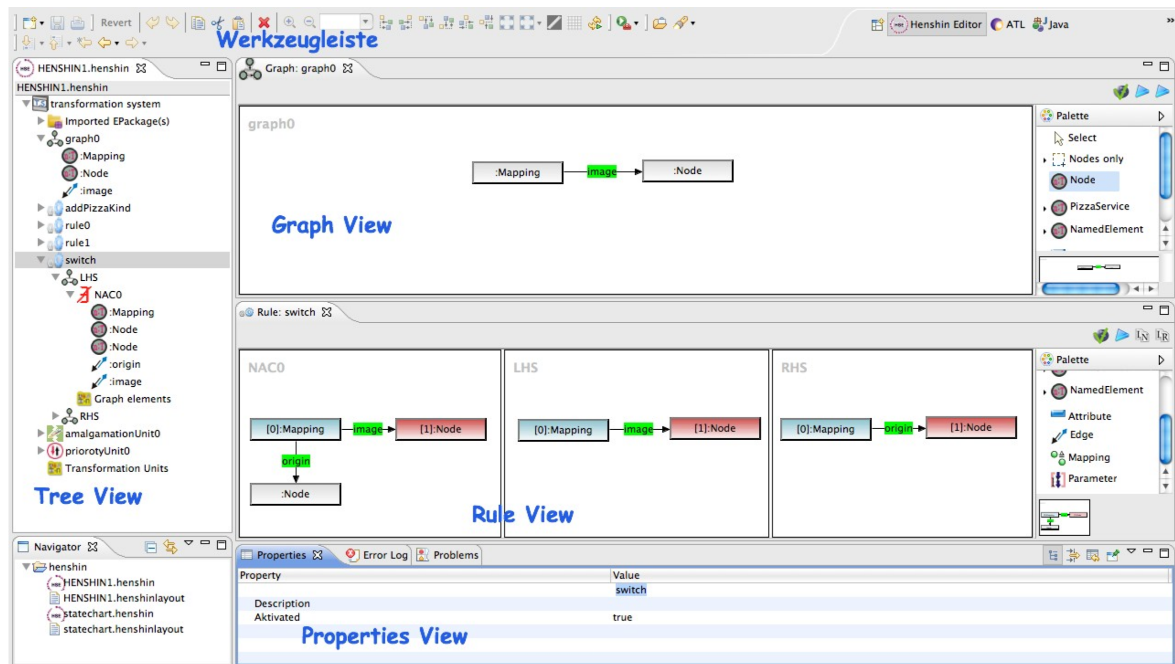


Abbildung 20: Henshin-Editor-Oberfläche

3.3 Funktionalitäten des Henshin-Editors im Ist-Stand

Der während des Projekts entwickelte Henshin-Editor umfasst folgende Funktionalitäten:

- Henshindateien erzeugen
- EPackages (.ecore) importieren und löschen
- Graphen erstellen, umbenennen, löschen und validieren
  - Knoten erstellen, umbenennen und löschen
    - Attribute erstellen und löschen
    - Attributwerte ändern
  - Kanten erstellen und löschen
- Regeln erstellen, umbenennen, löschen, validieren und ausführen
  - Parametern erstellen
  - Negative Anwendungsbedingungen (*NAC*) definieren
  - Mapping zwischen *LHS*- und *RHS*-Knoten erstellen
  - Mapping zwischen *LHS*- und *NAC*-Knoten erstellen

- Kopieren von *LHS* nach *RHS*
- Kopieren von *LHS* nach *NAC*

3.4 Anwendungsbeispiel einer Graphtransformation

Als Anwendungsbeispiel dient ein imaginärer Pizza-Service.

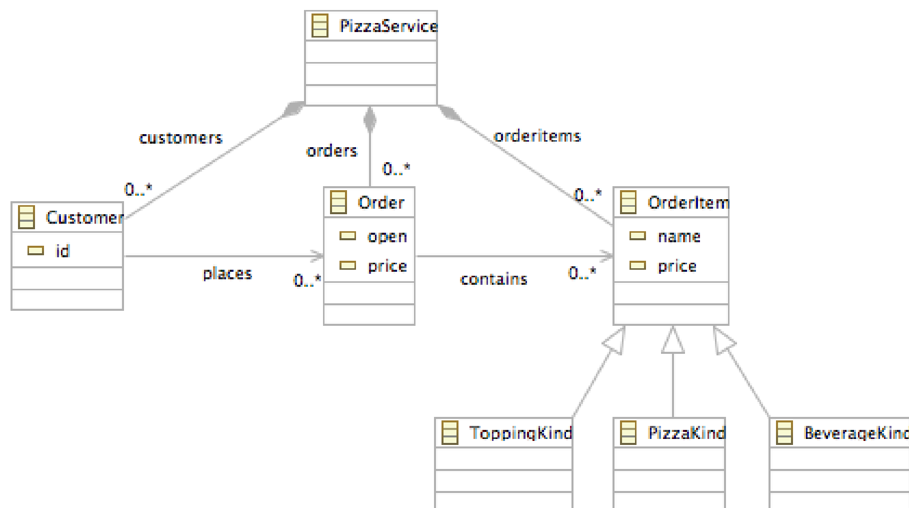


Abbildung 21: Pizza-Service als EMF-Modell

Es soll eine Regel *createOrder* ausgeführt werden, die einem *Customer* eine offene *Order* mit folgender Anwendungsbedingung zuweist: „Der *Customer* hat noch keine *Order*."

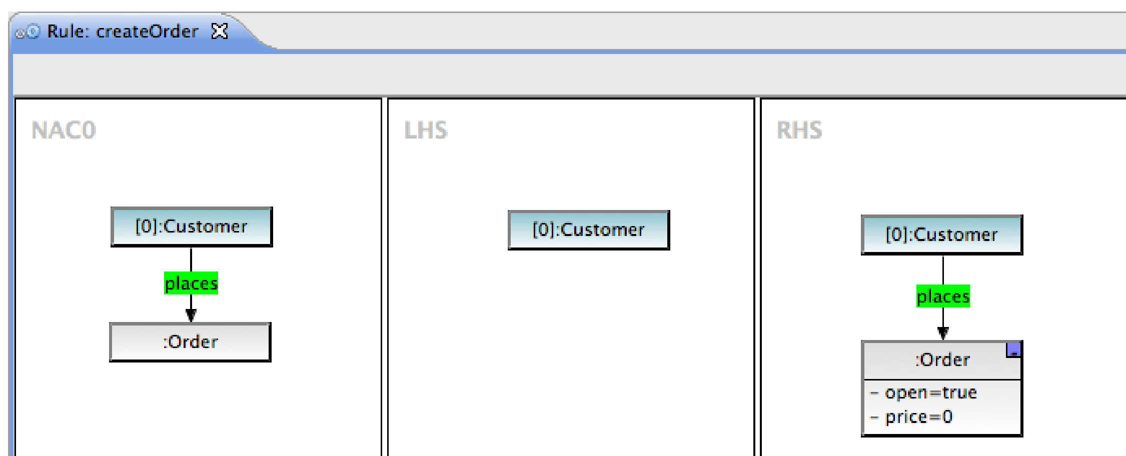


Abbildung 22: Regel *createOrder*

Die Anwendung der Regel *createOrder* auf *G2* ist verboten, weil das *NAC*-Abbild in *G2* existiert. In *G1* dagegen kann die Regel *createOrder* mit dem unten abgebildeten Zielgraphen erfolgreich ausgeführt werden.

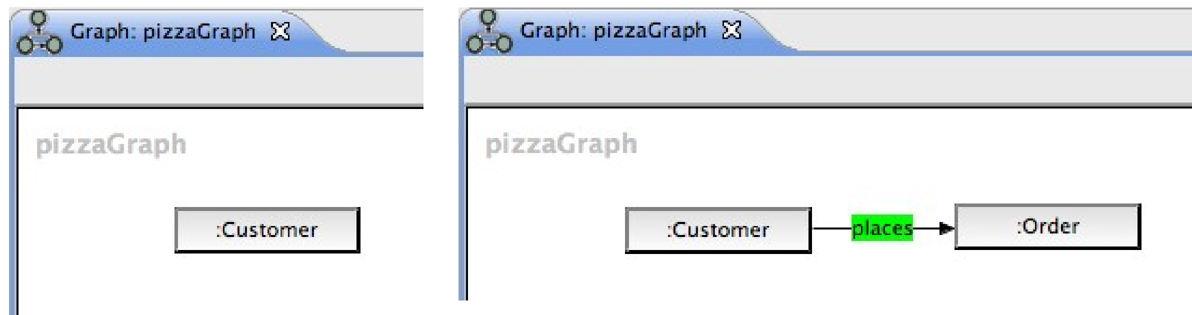


Abbildung 23: Arbeitsgraphen *G1* (links) und *G2* (rechts)

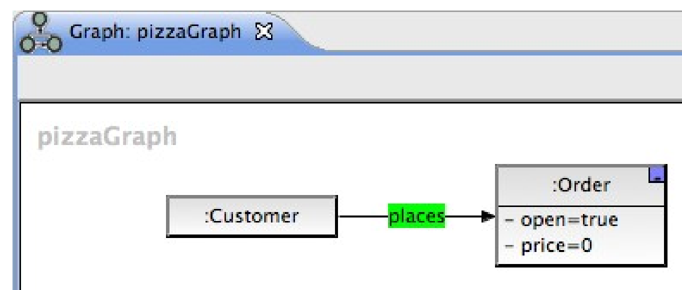


Abbildung 24: Arbeitsgraph *G1* nach der Regelausführung

4 Anforderungen

Dieses Kapitel behandelt die Anforderungen, die an die Erweiterung des Henshin-Editors gestellt werden. Zuerst werden einige wichtige Begriffe bezüglich des Arbeitsthemas erläutert. Danach werden Probleme erörtert, die ohne diese Arbeit auftreten könnten. Anschließend wird der Begriff *Anforderung* näher betrachtet und eine Methode vorgestellt, mit deren Hilfe Anforderungen eindeutig formuliert werden können. Abschließend werden die grundsätzlichen Ziele dieser Arbeit vorgestellt und die Anforderungen konkret formuliert.

4.1 Anwendungsbedingungen (*Application Condition, AC*)

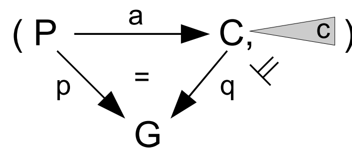
Während der ersten Entwicklung des Henshin-Editors im Rahmen des Visuelle-Sprachen-Projekts spielten negative Anwendungsbedingungen (*NAC*s) für die Graphtransformationsregeln bereits eine große Rolle. Sie können verwendet werden, um eine Regelanwendung zu verbieten, wenn eine bestimmte Struktur im Arbeitsgraphen vorgefunden wird.

Im Allgemeinen sollen Anwendungsbedingungen einen Mechanismus bieten, Regelanwendungen bei Graphtransformationen zu steuern. Darüber hinaus haben sie noch weitere wichtige Eigenschaften, die der Henshin-Editor im Rahmen dieser Arbeit berücksichtigen soll:

- Eine positive Anwendungsbedingung (*PAC*) wird definiert, um die Anwendung einer Regel nur dann zu erlauben, wenn der Arbeitsgraph eine bestimmte Struktur enthält.
- Anwendungsbedingungen können geschachtelt werden. Dies bedeutet, dass eine Anwendungsbedingung über eine andere Anwendungsbedingung definiert werden kann.
- Anwendungsbedingungen können negiert (*NOT*) werden und mehrere Anwendungsbedingungen können mit Hilfe der logischen Operatoren *AND* und *OR* kombiniert werden.

4.1.1 Graphbedingung

Sei G ein Arbeitsgraph. Eine Graphbedingung *gc (engl. graph condition)* über G ist entweder *true* ($\exists(a)$ bzw. $\exists(a, true)$) oder $\exists(a, c)$, wobei $a : P \rightarrow C$ *(engl. P = premise, C = conclusion)* ein Graphmorphismus und c eine Bedingung (logischer Ausdruck) über den Graphen C ist. Logische Ausdrücke über Bedingungen über P werden auch Bedingungen über P genannt. *[BESW10, EHL10b]*



Eine Anwendungsbedingung *ac* ist eine Graphbedingung über *LHS*.

4.1.2 Erfüllbarkeit einer Bedingung

Jeder Morphismus $p : P \rightarrow G$ erfüllt die Bedingung $ac = true$ (geschrieben $p \models ac$).

Ein Morphismus $p : P \rightarrow G$ erfüllt die Bedingung $ac = \exists(a, c)$, wenn ein injektiver Morphismus $q : C \rightarrow G$ existiert, so dass $q \circ a = p$ und $q \models c$ gilt. *[BESW10, EHL10b]*

Eine Transformationsregel kann nur ausgeführt werden, wenn die Anwendungsbedingung erfüllt ist (*true*).

4.1.3 Anwendungsbeispiel

Als Anwendungsbeispiel von Anwendungsbedingungen mit erweiterten Eigenschaften dient ein imaginärer Pizza-Service (siehe Abbildung 21, S. 20). Es soll eine Regel *gratisCola* definiert werden, bei der ein Kunde (*Customer*) eine Cola kostenlos bekommt, wenn eine der beiden folgenden Anwendungsbedingungen erfüllt ist:

1. Der Kunde bestellt *Menue1*: Zwei Pizzen, ein Extrabelag und ein Getränk.
2. Der Kunde bestellt *Menue2*: Drei Pizzen ohne Extrabelag (*!Extrabelag*).

Abbildung 25 – 28 zeigen die Darstellung der Regel *gratisCola* und ihre geschachtelten Anwendungsbedingungen im erweiterten Henshin-Editor.

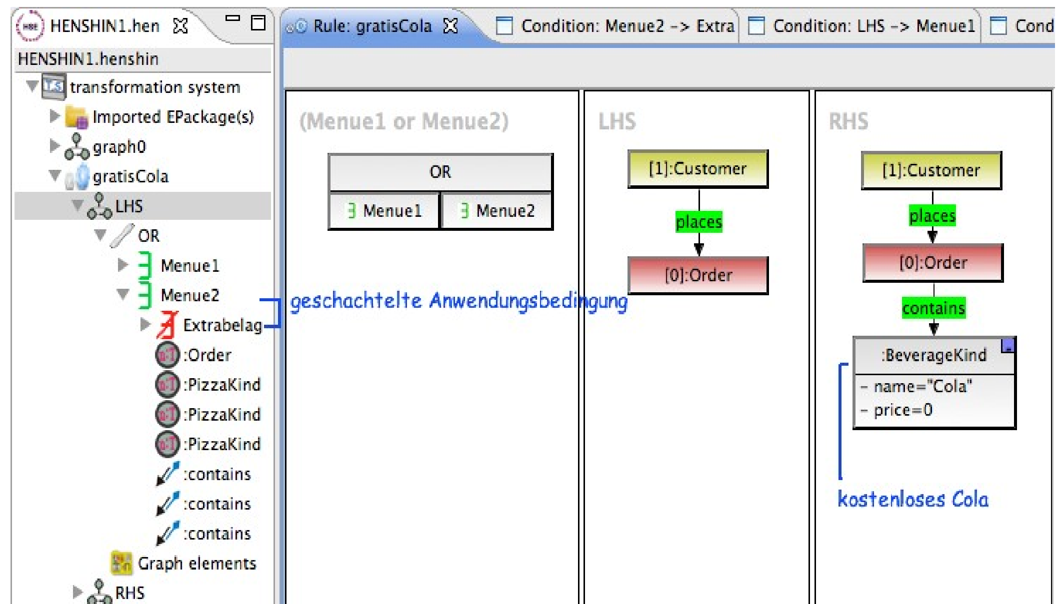


Abbildung 25: Die Regel *gratisCola*

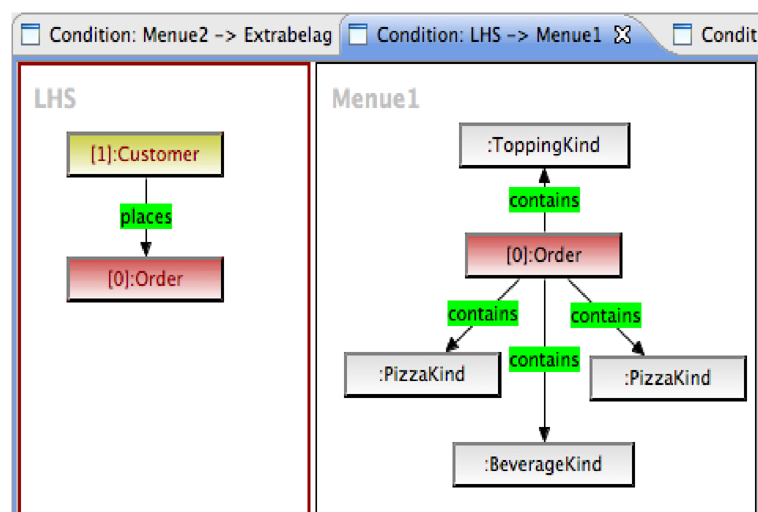


Abbildung 26: Die Anwendungsbedingung *Menue1*

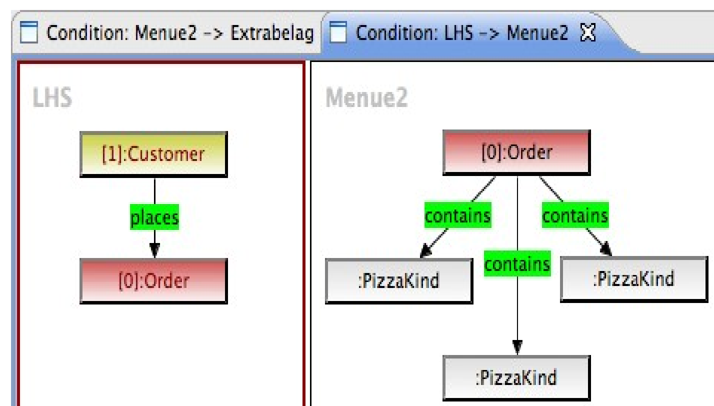


Abbildung 27: Die Anwendungsbedingung *Menue2*

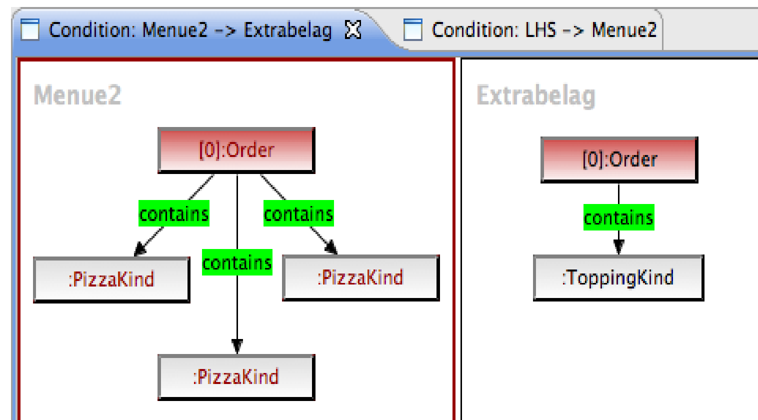


Abbildung 28: Die Anwendungsbedingung *Extrabelag*

4.2 Amalgamierte Regeln

Das Wort *„Amalgam"* stammt aus dem griechischen und bedeutet *„weich"*. „Ein Amalgam ist in der Chemie eine Legierung aus Quecksilber. Als Amalgam werden auch Vermischungen anderer Stoffe bezeichnet, die nicht ohne weiteres umkehrbar sind, z.B. Legierungen mehrerer Metalle." *[WP-AMAL10]*

Im Rahmen dieser Arbeit wird das Wort „Amalgamation" im Zusammenhang mit der Transformationsregel verwendet. Amalgamation-Units sind eine besondere Form von Transformationsregeln. Mit ihnen kann ein zu transformierender Graphteil definiert werden, dessen Anzahl des Auftretens im Arbeitsgraphen variabel ist. Dies bedeutet, dass die Regel auf alle gefundenen Matches angewendet wird.

Eine Amalgamation-Unit besteht aus einer Kernregel und mehreren Multiregeln (siehe Abbildung 29). In der Kernregel wird eine Regel mit ihren Bestandteilen, wie im Abschnitt 2.2.1 beschrieben, definiert. Die LHS- und RHS-Graphen der Kernregel sind Teilgraphen der jeweiligen LHS- und RHS-Graphen einer Multiregel. Zusätzlich enthält jede Multiregel Multiobjekte, auf deren Matches im Arbeitsgraphen die amalgamierte Regel parallel angewendet wird.

Aus einer Amalgamation-Unit und einem Graphen G kann eine amalgamierte Regel konstruiert werden. Eine amalgamierte Regel ist eine normale Transformationsregel, die den Effekt der Anwendung der Amalgamation-Unit beschreibt. Abbildung 30 zeigt die amalgamierte Regel für die Amalgamation-Unit in Abbildung 29 und den Graphen G in Abbildung 31.

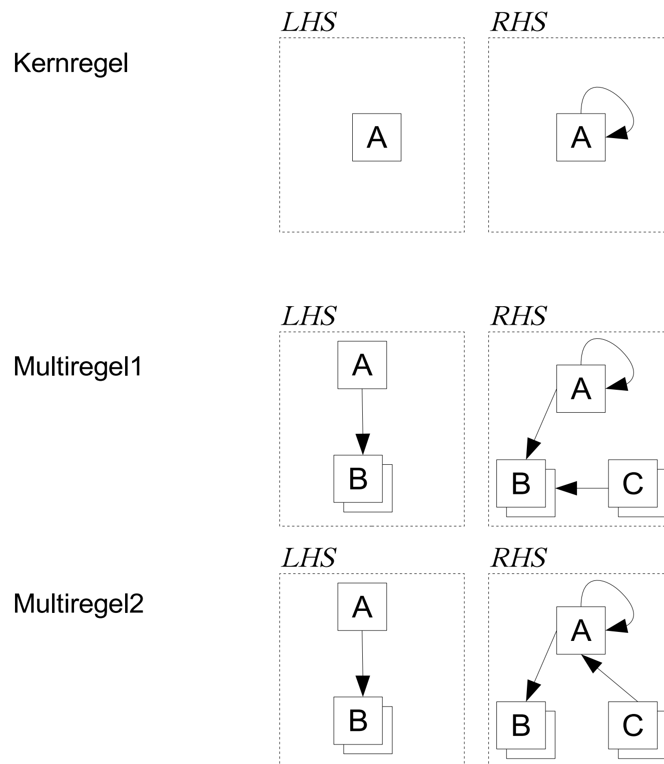


Abbildung 29: Amalgamation-Unit

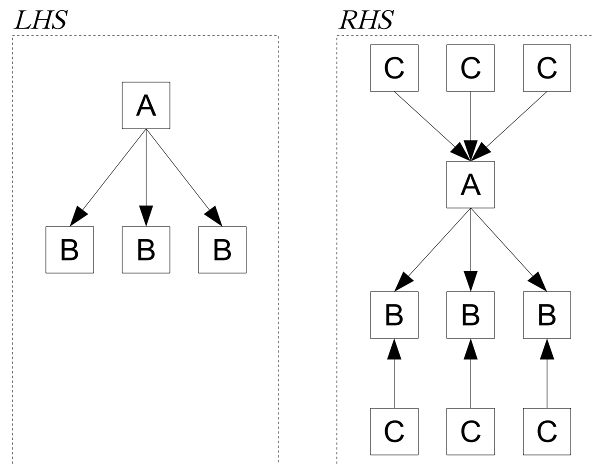


Abbildung 30: Amalgamierte Regel

In der Abbildung 31 wird der Graph G vor und nach der Anwendung der amalgamierten Regel dargestellt. Die braune Kante wurde durch die Anwendung der Kernregel eingefügt, die blauen Knoten und Kanten von der 1. und die grünen Knoten und Kanten von der 2. Multiregel. Es werden jeweils 3 C-Knoten von der amalgamierten Regel eingefügt, weil es 3 B-Knoten gibt, auf die die Multiobjekte der beiden *LHS*s von der 1. und der 2. Multiregel gematcht werden können.

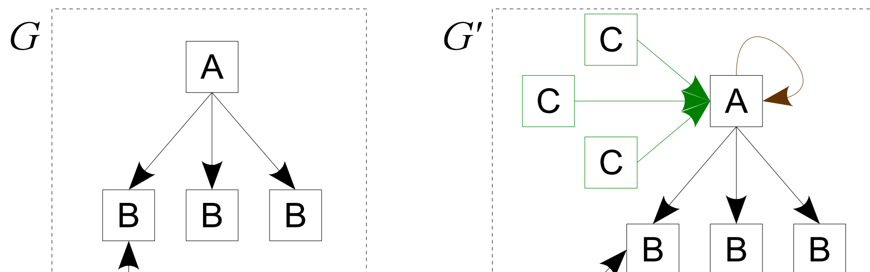


Abbildung 31: Graphen G vor und G' nach der Anwendung der oben definierten amalgamierten Regel

4.3 Potentielle Probleme ohne diese Arbeit

Die Erweiterung des existierenden Henshin-Editors ist notwendig, um die Mächtigkeit der Sprache Henshin vollständig nutzen zu können. Zur Zeit können Graphtransformationsregeln nur mit den folgenden Einschränkungen definiert werden:

- Es ist nicht möglich, mehrere gleiche Graphteile in einem Arbeitsgraphen mit einer Regel zu ändern (*AmalgamationUnit*).

  Wenn die Anzahl der Matches bekannt ist, kann das Problem umgangen werden, indem eine Regel mit der exakten Anzahl der zu transformierenden Graphteile in *LHS* und *RHS* definiert wird. Üblicherweise ist die Anzahl der Matches aber nicht bekannt.

- Es ist nicht möglich, die Regelanwendung auf die Existenz einer definierten Struktur in einer Anwendungsbedingung zu begrenzen (*PAC*).

- Es ist nicht möglich, Anwendungsbedingungen mit einem Disjunktor (logisches ODER) zu verknüpfen (*OR*).

  Dieses Problem kann umgangen werden, indem zwei Regeln mit dem gleichen *LHS*- und *RHS*-Graphen definiert werden. Die erste Regel enthält die erste Disjunkte und die zweite Regel die zweite. Ein erfolgreiches Ausführen der 1. oder der 2. Regel bedeutet, dass die disjunkten Anwendungsbedingungen erfüllt sind.

- Es ist nicht möglich, eine ganze Formel (Gesamtheit aller Anwendungsbedingungen) zu negieren (*NOT*).

4.4 Anforderung - Begriffsklärung und Vorgehensweise

Anforderungen beschreiben im Kontext der Informatik Leistungsmerkmale von Software. Der Rational Unified Process definiert Anforderungen folgendermassen: „A description of a condition or capability of a system; either derived directly from user needs or stated in a contract, standard, specification or other formally imposed document.“ *[Kru00]*

Alle im Abschnitt 4.5 beschrieben Anforderungen wurden als mustergültige Anforderungen formuliert. Mustergültige Anforderungen sind nach *[RD01]* ein „schablonenbasierter Weg zur Konstruktion und Qualitätssicherung von eindeutigen, vollständigen und testbaren Anforderungen“. Mit Hilfe mustergültiger Anforderungen wird versucht, die Vorteile der musterbasierten Herangehensweise auf den Bereich der Anforderungsanalyse zu übertragen.

Anforderungen werden bei dieser Methode nicht mehr frei formuliert, sondern nach bestimmten Regeln konstruiert, wodurch sie eine einheitliche Struktur erhalten. Man erhofft sich davon präzise formulierte Anforderungen, die von allen Beteiligten leicht zu verstehen sind. Typische Formulierungsfehler sollen mit dieser Technik möglichst ausgeschlossen werden, z.B. Sätze, bei denen nicht klar ist, von wem die Funktionalität erwartet wird.

Abbildung 32 zeigt das Prinzip einer Anforderungsschablone nach *[RD01]*.

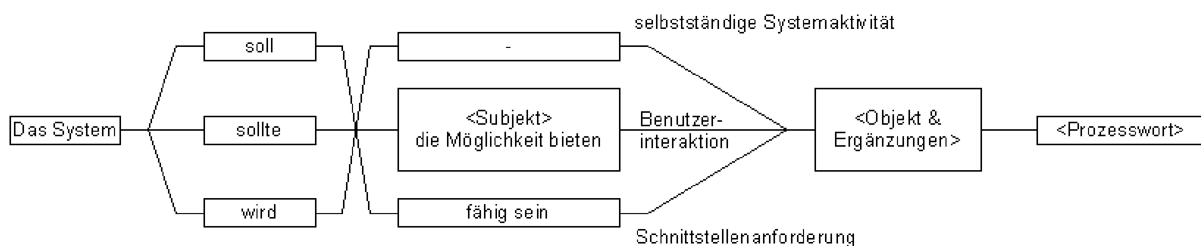


Abbildung 32: Prinzip einer Anforderungsschablone.

Um eine bessere Übersichtlichkeit zu erreichen, wird jede Anforderung in einer einheitlichen Tabelle mit grauem Hintergrund beschrieben. Die Tabelle enthält die Nummer und den Namen der Anforderung, einen mustergültigen Anforderungstext und eine Erläuterung. Alle Anforderungen zusammen bilden den Anforderungskatalog.

4.5 Anforderungen an den erweiterten Henshin-Editor

In diesem Abschnitt werden die Anforderungen an den erweiterten Henshin-Editor beschrieben. Die Anforderungen sind von der Projektveranstalterin, Dr. Claudia Ermel, und von anderen Mitarbeitern des Fachbereichs TFS festgelegt worden. Sie wurden als Themen der Abschlussarbeiten am Ende des Visuelle-Sprachen-Projekts im Wintersemester 2009/2010 vorgestellt. Grundsätzliches Ziel ist es, den Henshin-Editor so zu erweitern, dass

- komplexe und verschachtelte Ausdrücke über Graphen definiert und editiert werden können und
- Amalgamation-Units definiert und editiert werden können.

4.5.1 Anforderungen mit Nutzerinteraktionen

Dieser Abschnitt stellt die Anforderungen dar, bei denen Nutzerinteraktionen mit dem Henshin-Editor notwendig sind.

Anforderung 1:	***/Verschachtelte Anwendungsbedingungen/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, verschachtelte Anwendungsbedingungen für eine Regel definieren zu können.
Erläuterung	Es soll möglich sein, dass der Nutzer sowohl für LHS-Graphen als auch für einen AC-Graphen eine Anwendungsbedingung definiert.

Anforderung 2:	***/Komplexe Anwendungsbedingungen/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, komplexe Anwendungsbedingungen zu definieren.
Erläuterung	Eine Anwendungsbedingung lässt sich nicht nur mit NAC definieren, sondern auch mit PAC und den logischen Operatoren NOT, AND und OR. Logische Operatoren können wiederum selbst NAC, PAC oder logische Operatoren enthalten.

Anforderung 3:	***/NAC als Standard/***
Anforderungstext	Der Henshin-Editor soll beim Erstellen einer einzelnen *Application Condition* eine *NAC* als Standard in dem markierten Graphen (Prämisse) eingefügen.
Erläuterung	Bei Anwendungsbedingungen tritt öfter der Fall auf, dass ein bestimmtes Abbild der *Application Condition* nicht in dem Arbeitsgraphen zu finden ist. Aus diesem Grund soll *NAC* als Standardanwendungsbedingung verwendet werden.

Anforderung 4:	***/Negated-Wert ändern/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer erlauben, den *negated*-Wert einer *Application Condition* nachträglich zu ändern.
Erläuterung	Es gibt zwei Typen von *Application Conditions*: Negative (*NAC*) und positive (*PAC*). Eine *NAC* hat den *negated*-Wert *true*, bei einer *PAC* ist dieser *false*. Eine *NAC* soll nachträglich zu einer *PAC* geändert werden können und umgekehrt.

Anforderung 5:	***/Application Condition einfügen/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, einer existierenden Anwendungsbedingung eine *Application Condition* hinzufügen zu können.
Erläuterung	Diese Funktionalität betrifft nur *LHS*- und *AC*-Graphen mit einer definierten Anwendungsbedingung. In diesem Fall soll immer möglich sein, eine neue *Application Condition* hinzuzufügen.

Anforderung 6:	***/Tauschen von binären Formeln/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, eine erstellte *AND*-Formel mit einer *OR*-Formel zu tauschen und umgekehrt.
Erläuterung	Sowohl *AND* als auch *OR* sind logische Operatoren mit zwei Operanden. Daher soll es möglich sein, die beiden Operatoren zu tauschen.

Anforderung 7:	***/Erstellen von Amalgamation-Units/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, Amalgamation-Units zu definieren.
Erläuterung	Die Wurzel *transformation system* (siehe 3.2.1, S. 15) kann jetzt Amalgamation-Units beinhalten. Daher soll es möglich sein, Amalgamation-Units in das Transformationssystem einzufügen.

Anforderung 8:	***/Kernregel und Multiregeln definieren/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, einer Amalgamation-Units eine Kernregel und mehrere Multiregeln zu definieren.
Erläuterung	Eine Amalgamation-Unit besteht aus einer Kernregel und mehrerer Multiregeln (siehe 4.2, S. 26). Mit dem Henshin-Editor soll der Nutzer diese Regeln definieren können.

Anforderung 9:	***/Korrekte Amalgamation-Unit/***
Anforderungstext	Der Henshin-Editor soll für die Erstellung korrekter Amalgamation-Units sorgen.
Erläuterung	Es soll sichergestellt werden, dass eine Amalgamation-Unit aus einer Kernregel und (eventuell) mehreren Multiregeln besteht (siehe 4.2, S. 26). Eine Amalgamation-Unit mit Multiregeln ohne Kernregel ist unzulässig.

Anforderung 10:	***/Abbild der Kernregel/***
Anforderungstext	Der Henshin-Editor soll das Abbild einer Kernregel in jeder Multiregel exakt darstellen.
Erläuterung	Da *LHS*- und *RHS*-Graphen der Kernregel Teilgraphen der jeweiligen *LHS*- und *RHS*-Graphen einer Multiregel sind, soll sichergestellt werden, dass das Abbild der Kernregel nach jeder Nutzeraktionen bezüglich einer Amalgamation-Unit in den Multiregeln vorgefunden wird.

Anforderung 11:	***/Kopieren von LHS nach RHS in Multiregeln/***
Anforderungstext	Der Henshin-Editor soll dem Nutzer die Möglichkeit bieten, Multiobjekte von *LHS* nach *RHS* zu kopieren.
Erläuterung	Die Multiobjekte in *LHS* und *RHS* sind oft identisch. Deswegen ist es hilfreich, wenn sie von *LHS* nach *RHS* kopiert werden können.

4.5.2 Anforderungen an die Darstellung von Objekten

Dieser Abschnitt beinhaltet Anforderungen, die mit der Darstellung von Objekten des erweiterten Henshin-Editors zu tun haben.

Anforderung 12:	***/Darstellung von Anwendungsbedingungen/***
Anforderungstext	Der Henshin-Editor soll die Anwendungsbedingungen in verschiedene Varianten darstellen.
Erläuterung	Wie im Abschnitt 4.2 (S. 28) beschrieben wurde, hat der Henshin-Editor mehrere Ansichten. Zum besseren Übersichtlichkeit sollen Anwendungsbedingungen in den entsprechenden Ansichten in verschiedene Varianten dargestellt werden.

Anforderung 13:	***/Erkennen von unvollständigen Anwendungsbedingungen/***
Anforderungstext	Der Henshin-Editor soll unvollständige Anwendungsbedingungen anders darstellen als vollständige.
Erläuterung	Die unterschiedliche Darstellung soll dem Nutzer helfen, unvollständige Anwendungsbedingungen leichter zu erkennen. Zu unvollständigen Anwendungsbedingungen gehören: • *NOT* ohne Operand und • *AND* bzw. *OR* ohne Operanden oder nur mit einem Operanden.

Anforderung 14:	***/Erkennen von PAC und NAC/***
Anforderungstext	Der Henshin-Editor soll positive und negative Anwendungsbedingungen unterschiedlich darstellen.
Erläuterung	Der Nutzer soll leicht erkennen können, ob eine *Application Condition* negiert ist (*NAC*) oder nicht (*PAC*).

Anforderung 15:	***/Visualisierung von Mappings zwischen Knoten/***
Anforderungstext	Der Henshin-Editor soll gemappte Knoten graphisch darstellen.
Erläuterung	Die Mappings zwischen den Knoten sollen für den Nutzer graphisch leicht zu erkennen sein. In der Regelansicht soll sofort erkennbar sein, welcher Knoten mit welchem gemappt ist.

Anforderung 16:	***/Darstellung von Amalgamation-Units/***
Anforderungstext	Der Henshin-Editor soll Amalgamation-Units in verschiedene Varianten darstellen.
Erläuterung	Wie im Abschnitt 3.2 (S. 15) beschrieben wurde, hat der Henshin-Editor mehrere Ansichten. Zur besseren Übersichtlichkeit sollen Amalgamation-Units in den verschiedenen Ansichten dargestellt werden.

Anforderung 17:	***/Darstellung von Kernregel und ihrer Multiregeln/***
Anforderungstext	Der Henshin-Editor soll die Kernregel und die dazu gehörigen Multiregeln darstellen.
Erläuterung	Wie im Abschnitt 4.2 (S. 28) beschrieben wurde, hat eine Amalgamation-Unit eine Kernregel und mehrere Multiregeln. Zur besseren Übersichtlichkeit sollen die Kernregel und ihre Multiregeln graphisch dargestellt werden, so dass erkennbar ist, dass sie zu einer Amalgamation-Unit gehören.

Anforderung 18:	***/Darstellung von Kern- und Multiknoten/***
Anforderungstext	Der Henshin-Editor soll die Kernknoten und die Multiknoten in den Multiregeln unterschiedlich darstellen.
Erläuterung	Die unterschiedliche Darstellung von Kern- und Multiknoten soll dem Nutzer helfen, leichter zu erkennen, welche Knoten nur einmal im Arbeitsgraphen vorkommen sollen und welche mehrmals vorkommen dürfen. Außerdem ist durch die unterschiedliche Darstellung leichter erkennbar, welche Knoten geändert werden können. Nur die Multiknoten dürfen im Multiregel-Editor editiert und gelöscht werden.

5 Lösungsansatz

In diesem Kapitel werden die gewählten Lösungsansätze vorgestellt und anhand der in Kapitel 4 beschriebenen Anforderungen bewertet. Abschließend werden alternative Lösungsansätze erörtert.

5.1 Vorstellung der gewählten Lösungsansätze

In diesem Abschnitt wird diskutiert, ob die gewählten Lösungsansätze als geeignet gelten können, um die in Kapitel 4 beschriebenen Anforderungen zu erfüllen. Zu diesem Zweck werden die im Abschnitt 4.5 formulierten mustergültigen Anforderungen einzeln überprüft.

Anforderung 1: ***/Verschachtelte Anwendungsbedingungen/***

Der ausgebaute Henshin-Editor ermöglicht eine Verschachtelung von Anwendungsbedingungen. Dies bedeutet, dass eine Anwendungsbedingung über eine andere Anwendungsbedingung definiert werden kann. Die Menüpunkte zum Erstellen von Anwendungsbedingungen sind sowohl im Kontextmenü einer ausgewählten Regel (Abbildung 33) bzw. eines ausgewählten LHS-Graphen (Abbildung 34) sichtbar als auch im Kontextmenü eines ausgewählten AC-Graphen (Abbildung 35) in der Baumansicht.

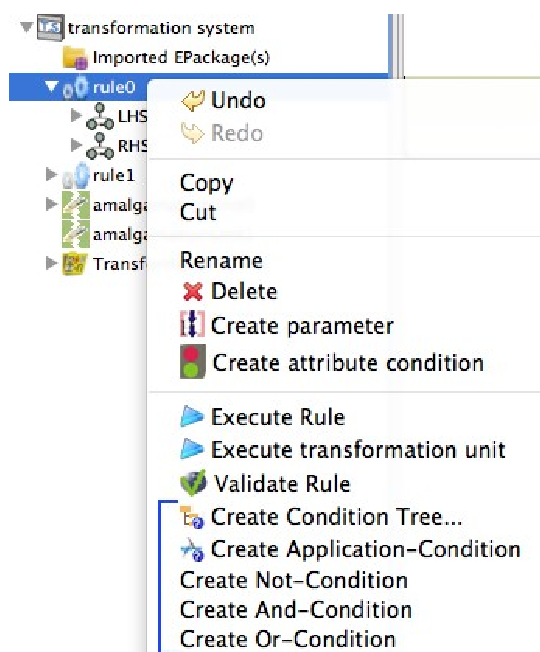


Abbildung 33: Kontextmenü einer Regel

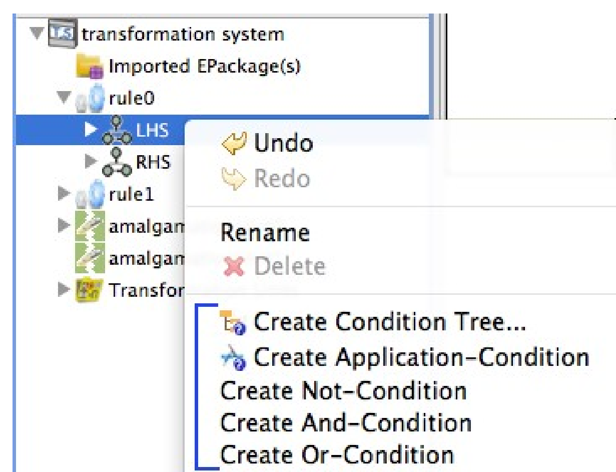


Abbildung 34: Kontextmenü eines LHS-Graphen

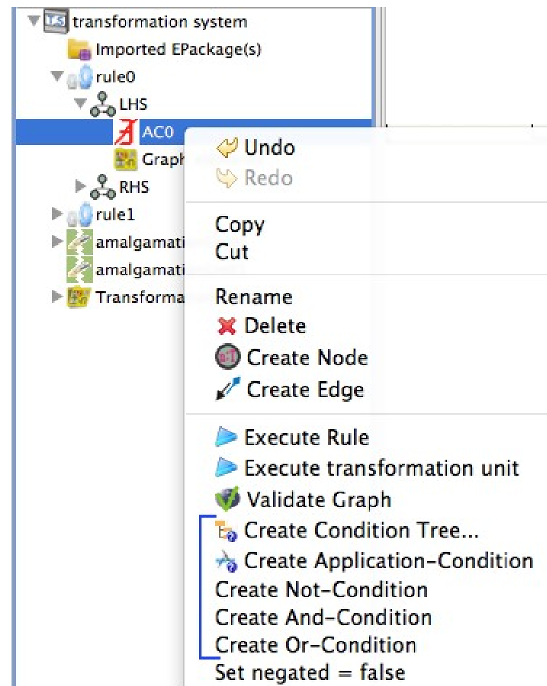


Abbildung 35: Kontextmenü eines AC-Graphen

Anforderung 2: ***/Komplexe Anwendungsbedingungen/***

Der ausgebaute Henshin-Editor ermöglicht nicht nur das Erzeugen von negativen Anwendungsbedingungen, sondern auch von positiven Anwendungsbedingungen und von den logischen Operatoren *NOT*, *AND* und *OR*.

Das Erstellen komplexer Anwendungsbedingungen erfolgt durch Auswahl einer der folgenden Menüpunkte im Kontextmenü einer Regel, eines *LHS*-Graphen oder eines *AC*-Graphen:

- *Create Condition Tree...*

  Eine Anwendungsbedingung kann in einem separaten Dialogfenster als Ganzes erstellt werden. Dies bedeutet, dass der Nutzer die Möglichkeit hat, eine vollständige komplexe Anwendungsbedingung zuerst im Dialogfenster (Abbildung 36) in der Baumdarstellung zusammenzubauen, bevor sie in einem Graphen hinzugefügt wird.

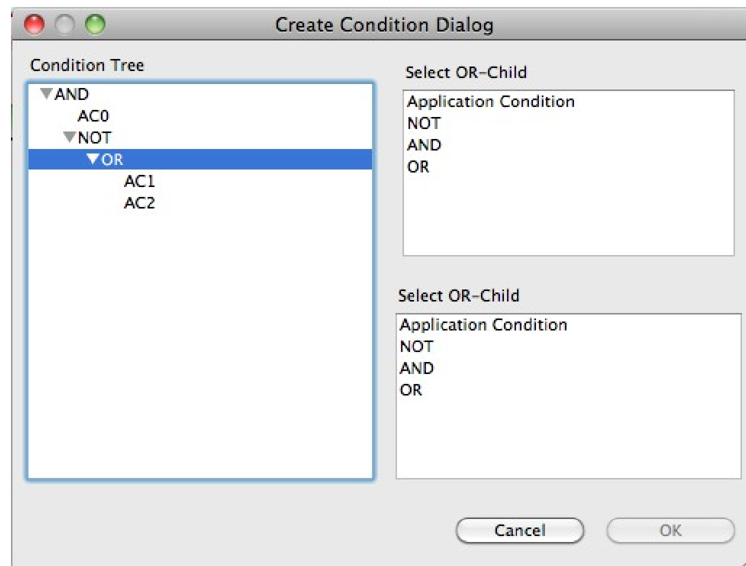


Abbildung 36: Dialogfenster zum Erstellen von (komplexen) Anwendungsbedingungen

- Eine Anwendungsbedingung kann schrittweise durch Auswahl einer der Menüpunkte *Create Application-Condition*, *Create Not-Condition*, *Create And-Condition* oder *Create Or-Condition* erstellt werden. Die ausgewählte Condition wird an den entsprechenden Graphen angehängt. In der Baumansicht ist dies durch einen neuen Eintrag unter dem ausgewählten Graphen zu erkennen.

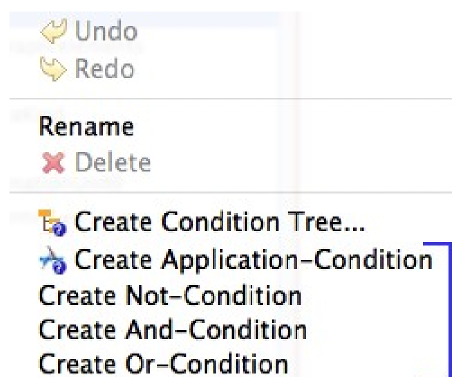


Abbildung 37: Menüpunkte zum schrittweisen Erstellen von (komplexen) Anwendungsbedingungen

Anforderung 3: ***/NAC als Standard/***

Wie weiter oben beschrieben, bietet der ausgebaute Henshin-Editor zwei Möglichkeiten, Anwendungsbedingungen zu erstellen: als Ganzes oder schrittweise. In beiden Varianten wird *NAC* als Standard eingefügt, wenn *Create Application-Condition* ausgewählt ist. Im Dialogfenster ist dies explizit durch ein markiertes *negated*-Feld zu erkennen (siehe Abbildung 38).

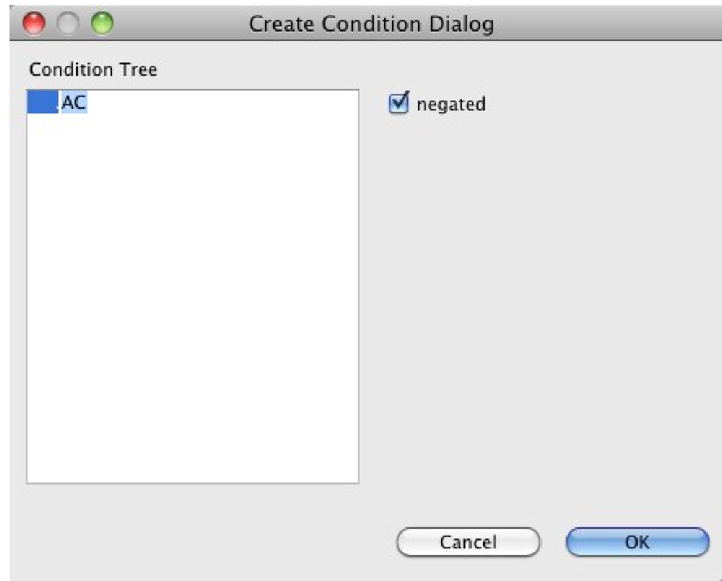


Abbildung 38: NAC als Standard im Dialogfenster

Anforderung 4: ***/Negated-Wert ändern/***

Der ausgebaute Henshin-Editor bietet dem Nutzer die Möglichkeit, den *negated*-Wert einer erstellten *Application Condition* zu ändern. Dadurch kann eine *NAC* zu *PAC* geändert werden und umgekehrt.

Eine markierte *NAC* in der Baumansicht hat den Menüpunkt *Set negated = false* in ihrem Kontextmenü, um sie zu einer *PAC* zu ändern. Umgekehrt hat eine *PAC* den Menüpunkt *Set negated = true*, um sie zu einer *NAC* zu ändern. Durch Auswählen dieses Menüpunktes ändert sich der *negated*-Wert (siehe Abbildung 39).

In der Properties-Ansicht hat eine markierte *Application Condition* entweder den *negated*-Wert *true (NAC)* oder *false (PAC)*. Durch Auswahl eines verfügbaren Wertes in der Kombinationsschaltfläche des *Negated*-Property ändert sich der *negated*-Wert (siehe Abbildung 40).

Anforderung 5: ***/Application Condition einfügen/***

Der Henshin-Editor bietet dem Nutzer die Möglichkeit, eine *Application Condition* einem *LHS*-Graphen bzw. einem *AC*-Graphen immer hinzuzufügen. Dies bedeutet, wenn der *LHS*- bzw. *AC*-Graph bereits über eine Anwendungsbedingung verfügt, werden die neue *Application Condition* und die existierende mit dem logischen Operator *AND* verbunden.

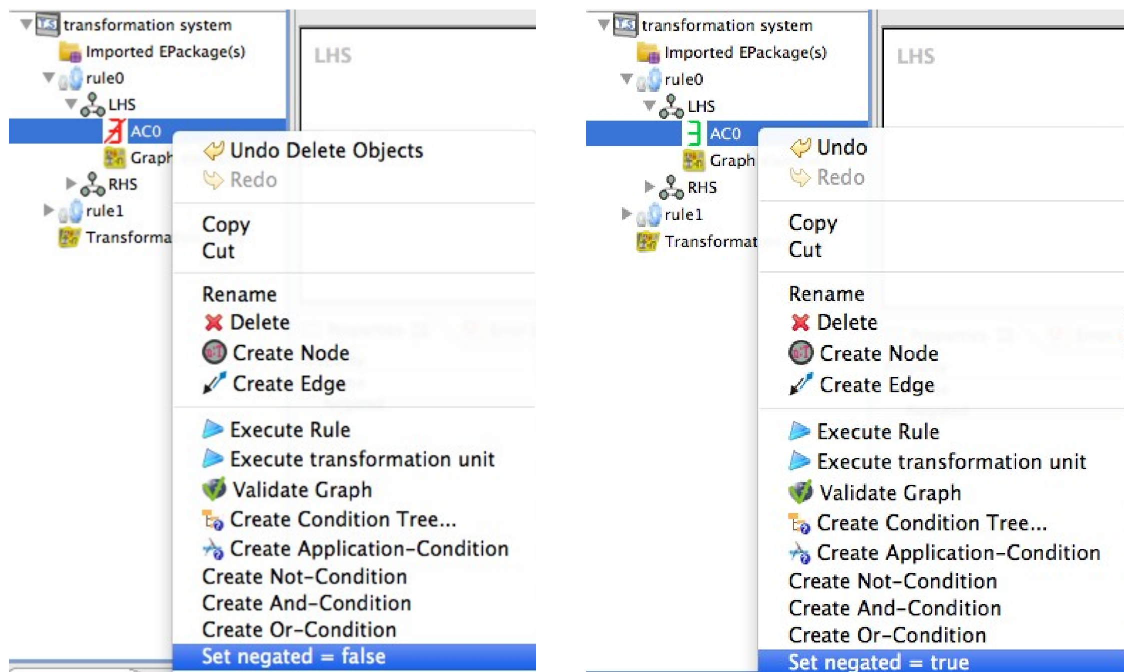


Abbildung 39: Ändern des negated-Wertes über das AC-Kontextmenü

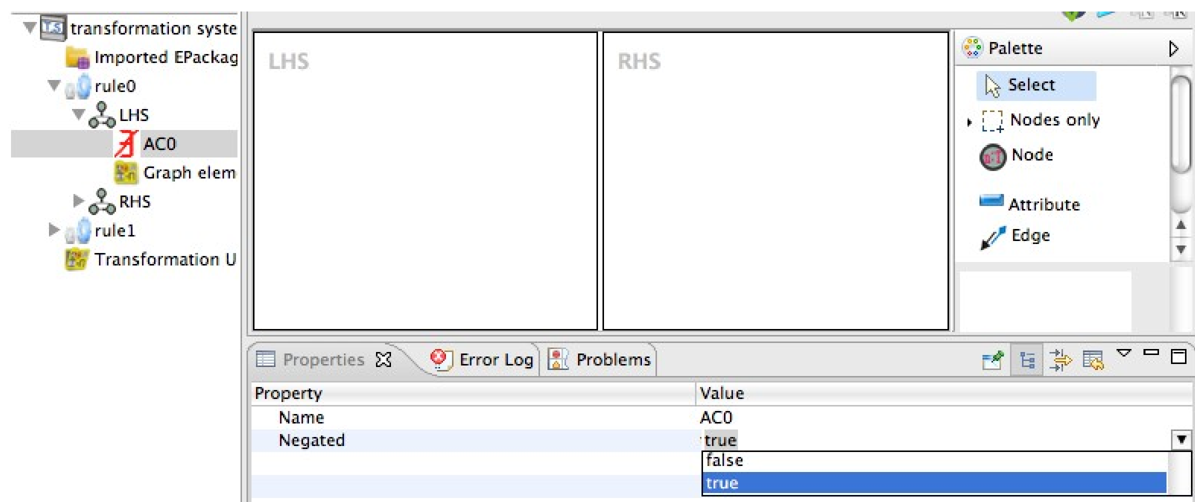


Abbildung 40: Ändern des negated-Wertes in der Properties-Ansicht

Das Hinzufügen einer *Application Condition* erfolgt sowohl über den Menüpunkt *Create Application-Condition* im Kontextmenü eines *LHS*- bzw. *AC*-Graphen in der Baumansicht, in den die *Application Condition* eingefügt werden soll, als auch im Kontextmenü einer Regel.

Anforderung 6: ***/Tauschen von binären Formeln/***

Der ausgebaute Henshin-Editor bietet dem Nutzer die Möglichkeit, eine binäre Formel *AND* mit einer binären Formel *OR* zu tauschen und umgekehrt. Beim Tauschen der Formeln werden die Operanden der alten Formel kopiert und an die neue Formel angehängt.

Das Tauschen von der binären Formel *AND* mit *OR* erfolgt durch Auswahl des Menüpunktes *Swap AND* → *OR* im *AND*-Kontextmenü in der Baumansicht und das Tauschen der binären Formel *OR* mit *AND* durch *Swap OR* → *AND* im *OR*-Kontextmenü (siehe Abbildung 41).

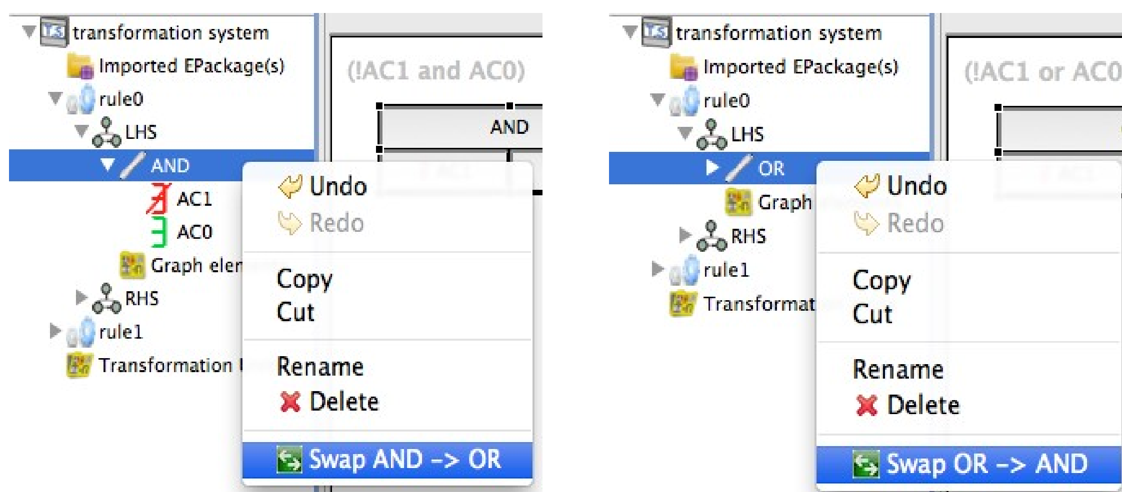


Abbildung 41: Tauschen von binären Formeln über das Kontextmenü

Anforderung 7: ***/Erstellen von Amalgamation-Units/***

Der ausgebaute Henshin-Editor bietet dem Nutzer die Möglichkeit, das Transformationssystem um Amalgamation-Units zu erweitern. Hierfür steht ein Menüpunkt *Create transformation unit with content* → *Create amalgamation unit* im Kontextmenü des Transformationssystems (Abbildung 42) oder im Kontextmenü des Containers *Transformation Units* (Abbildung 43) zur Verfügung.

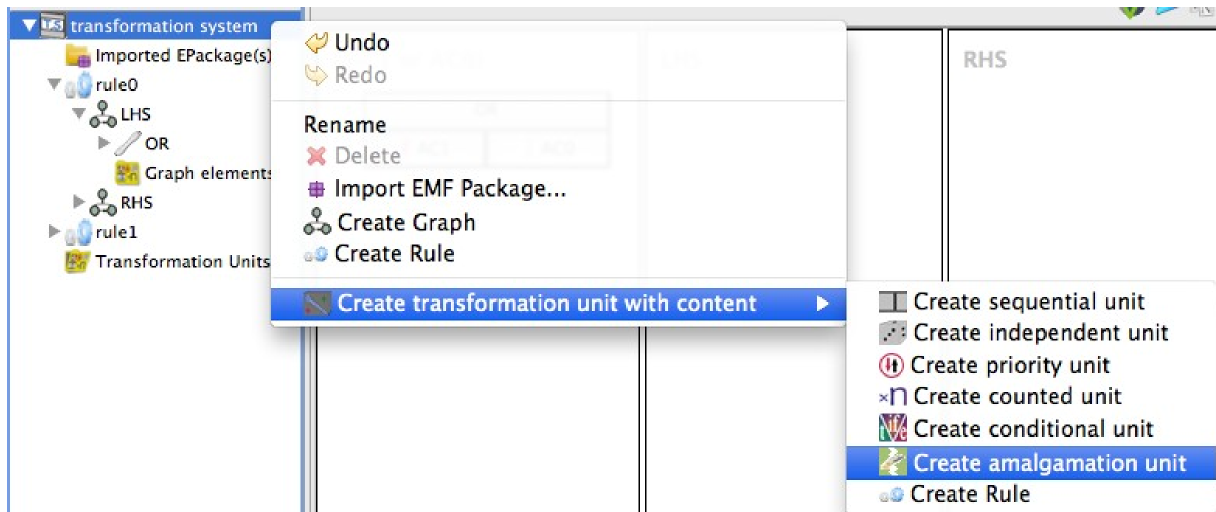


Abbildung 42: Erstellen einer Amalgamation-Unit über das Kontextmenü des Transformationssystem

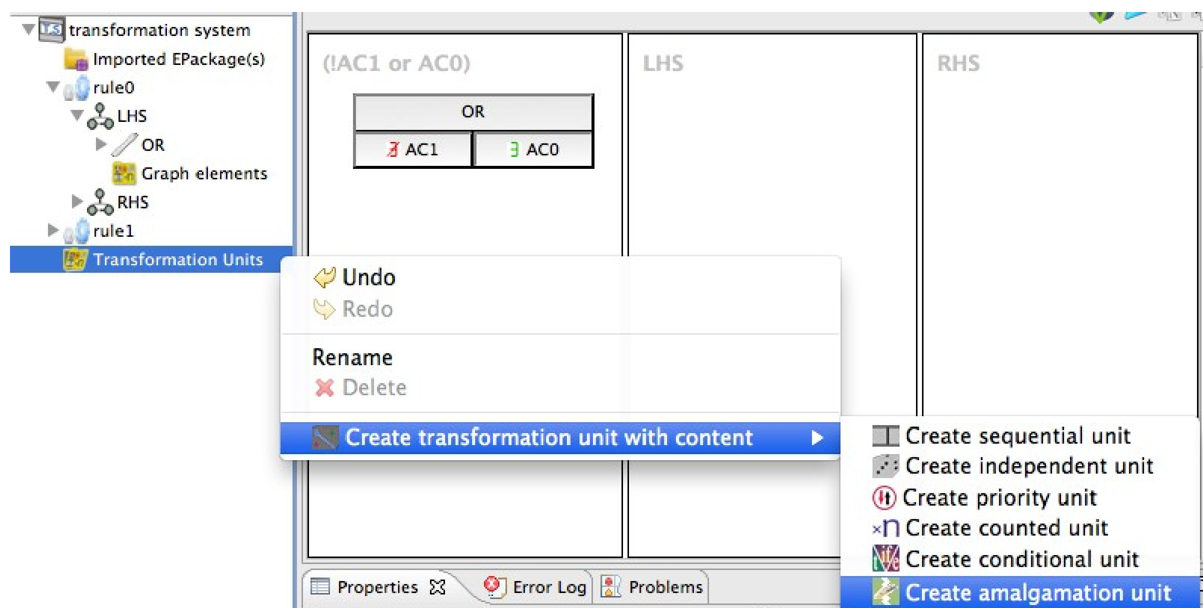


Abbildung 43: Erstellen einer Amalgamation-Unit über das Kontextmenü des Container *Transformation Units*

Anforderung 8: ***/Kernregel und Multiregeln definieren/***

Der ausgebaute Henshin-Editor gestattet es dem Nutzer, eine Kernregel und mehrere Multiregeln für jede existierende Amalgamation-Unit zu definieren.

Eine Kernregel kann über zwei Menüpunkte im Kontextmenü einer markierten Amalgamation-Unit definiert werden:

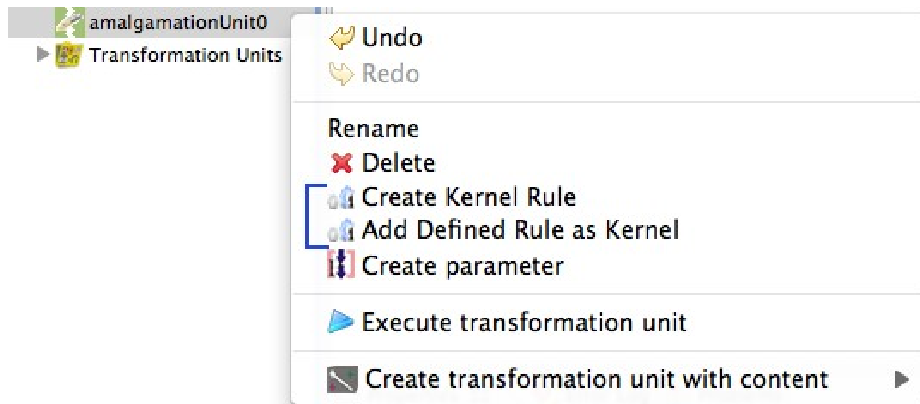


Abbildung 44: Definieren einer Kernregel über das Kontextmenü

- *Create Kernel Rule*

  Beim Auswählen dieses Menüpunktes wird eine neue Transformationsregel erstellt und in das Transformationssystem eingefügt. Diese Regel wird dann als Kernregel der markierten Amalgamation-Unit definiert.

- *Add Defined Rule as Kernel*

  Beim Auswählen dieses Menüpunktes wird eine existierende Transformationsregel als Kernregel definiert. Wenn nur eine Transformationsregel im Transformationssystem existiert, wird diese automatisch als Kernregel definiert. Andernfalls wird eine Liste mit den existierenden Transformationsregeln in einem Dialogfenster angezeigt.

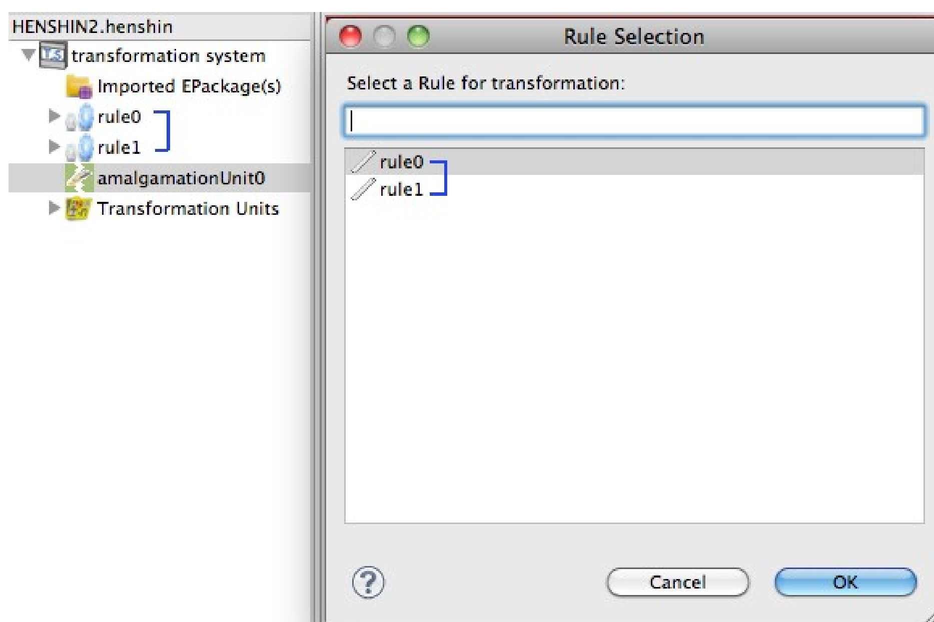


Abbildung 45: Dialogfenster zum Auswählen einer Regel

Durch Auswahl des Menüpunktes *Create Multi Rule* im Kontextmenü einer Amalgamation-Unit wird eine Transformationsregel erstellt und in das Transformationssystem eingefügt. Diese Regel wird dann als Multiregel der markierten Amalgamation-Unit definiert.

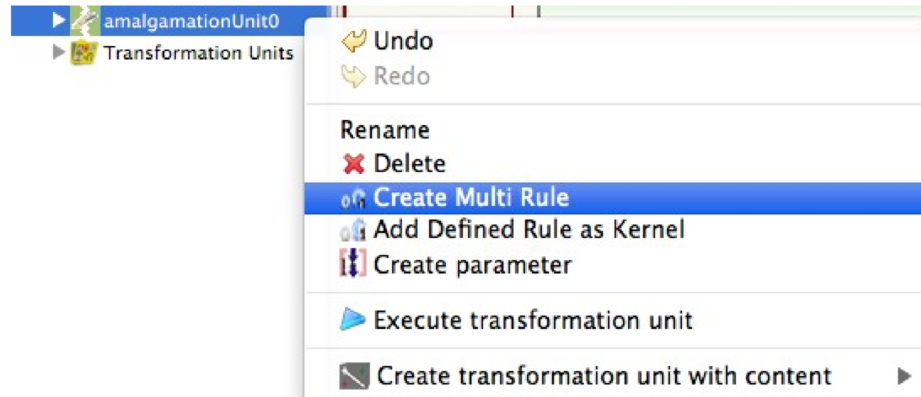


Abbildung 46: Definieren einer Multiregel über das Kontextmenü einer Amalgamation-Unit

Anforderung 9: ***/Korrekte Amalgamation-Unit/***

Der ausgebaute Henshin-Editor stellt sicher, dass Amalgamation-Units korrekt definiert werden. Dies bedeutet, dass sie nur aus einer Kernregel und aus (eventuell) mehreren Multiregeln bestehen. Das Sicherstellen korrekter Amalgamation-Units erfolgt durch Ausblenden unzulässiger Aktionen im Kontextmenü einer Amalgamation-Unit.

- Der Menüpunkt *Create Kernel Rule* ist nur sichtbar, wenn die Amalgamation-Unit noch keine Kernregel hat. Dies stellt sicher, dass eine Amalgamation-Unit nur aus einer Kernregel bestehen darf.
- Der Menüpunkt *Add Defined Rule as Kernel* ist nur sichtbar, wenn
    - das Transformationssystem mindestens eine Transformationsregel hat.

      Dies verhindert das Anzeigen eines Dialogfensters mit einer leeren Liste.
    - die Amalgamation-Unit noch keine Multiregel hat.

      Dies verhindert, dass der Inhalt von *LHS*- und *RHS*-Graphen in den Multiregeln verworfen wird.
- Der Menüpunkt *Create Multi Rule* ist nur sichtbar, wenn die Amalgamation-Unit bereits über eine Kernregel verfügt.

Dies verhindert den Zustand, dass eine Amalgamation-Unit nur aus Multiregeln besteht.

Anforderung 10: ***/Abbild der Kernregel/***

Der ausgebaute Henshin-Editor stellt sicher, dass das Abbild einer Kernregel in den jeweiligen Multiregeln nach jeder Nutzeraktion vorgefunden wird. Um dies zu erreichen, synchronisiert der Henshin-Editor automatisch jede Änderung von *LHS*- und *RHS*-Elementen einer Kernregel in die jeweiligen Multiregeln oder er verbietet Änderungen in Multiregeln, die zu einem inkonsistenten Zustand führen würden.

Die Sicherstellung betrifft folgende Nutzeraktionen:

- Erstellen von Multiregeln

  Beim Erstellen einer Multiregel werden alle Elemente (Knoten, Kanten, Attribute, Mappings, Parameter) der Kernregel kopiert und in die Multiregel eingefügt.

- Einfügen/Löschen eines Elements in der Kernregel

  Jede Änderung an der Kernregel wird auch an ihren Multiregeln durchgeführt. Diese automatische Synchronisation sorgt dafür, dass die Kernelemente in den Multiregeln konsistent bleiben.

- Kein Ändern/Löschen von Kernelementen in Multiregeln

  Der Henshin-Editor bietet dem Nutzer keine Möglichkeit, Kernelemente in Multiregeln zu editieren oder zu löschen, um inkonsistente Zustände in der Multiregel zu vermeiden.

Anforderung 11: ***/Kopieren von LHS nach RHS in Multiregeln/***

In der Ansicht einer Multiregel bietet der ausgebaute Henshin-Editor dem Nutzer die Möglichkeit, mit einem Klick die Multiobjekte mit all ihren Eigenschaften von *LHS* nach *RHS* zu kopieren (siehe Abbildung 47). Diese Kopieraktion erstellt ebenfalls die Mapping zwischen den Knoten automatisch. Die Kernelemente in dem *LHS*-Graphen werden nicht mitkopiert, weil dies einen inkonsistenten Zustand in der Multiregel zur Folge hätte.

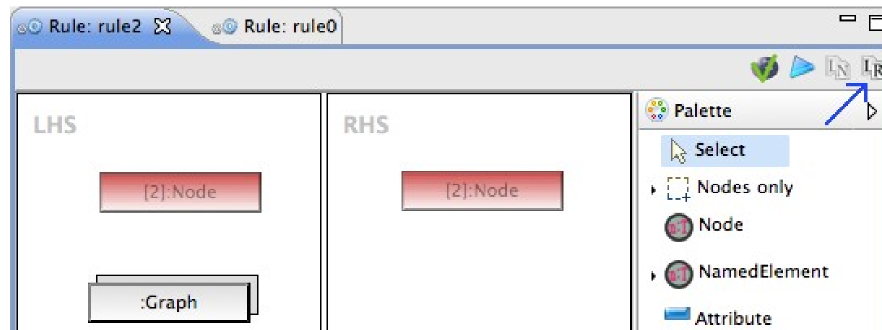


Abbildung 47: Schaltfläche zum Kopieren von Multiobjekte

Anforderung 12: ***/Darstellung von Anwendungsbedingungen/***

Der ausgebaute Henshin-Editor stellt Anwendungsbedingungen in verschiedenen Formen dar.

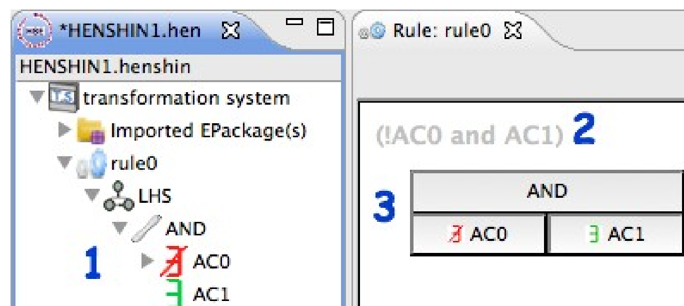


Abbildung 48: Darstellung von Anwendungsbedingungen

1. als Baum

   In der Baumansicht werden Anwendungsbedingungen als innere Einträge unter einem *LHS*-Eintrag dargestellt. Geschachtelte Anwendungsbedingungen werden unter einem *AC*-Eintrag als innere Einträge angezeigt.

2. als Text

   In der Regelansicht werden Anwendungsbedingungen erster Ebene als Text (logische Formel) dargestellt. Geschachtelte Anwendungsbedingungen können nicht sofort erkannt werden.

3. als Figur

   In der Regelansicht werden Anwendungsbedingungen erster Ebene durch einen Block dargestellt, der sich selbst aus mehreren anderen Blöcken zusammensetzt. Jeder Block repräsentiert dabei entweder einen Operator (*AND*, *OR*, *NOT*) oder eine *Application Condition*.

Bei einem Operator erscheinen die Operanden unterhalb des Operators. Bei Operatoren mit einem Operanden (*NOT*) hat der Block des Operanden dieselbe Breite wie der des Operators. Bei Operatoren mit zwei Operanden (*AND*, *OR*) haben die beiden Blöcken der Operanden die halbe Breite des Operatorkastens. In Abbildung 48 hat der Operator *AND* die Operanden *AC0* und *AC1*.

Mit einem Doppelklick auf einem *AC*-Blöcken öffnet sich eine Bedingungsansicht (*Condition View*), auf deren linker Seite die Prämisse und auf deren rechter Seite die Bedingung dargestellt sind. Hat die Prämisse nur eine *Application Condition* als Anwendungsbedingung, wird diese nicht als Blockfigur präsentiert, sondern direkt als *AC*-Graph.

Anforderung 13: ***/Erkennen von unvollständigen Anwendungsbedingungen/***

Im ausgebauten Henshin-Editor werden unvollständige Anwendungsbedingungen sowohl in der Baumansicht als auch in der Blockfigur rot beschriftet. Dadurch ist für den Nutzer leicht erkennbar, dass weitere Anwendungsbedingungen definiert werden müssen, um die Anwendungsbedingung zu vervollständigen. Ein weiteres Kennzeichen einer unvollständigen Anwendungsbedingung ist das Fehlen der Textdarstellung in der Regelansicht. In Abbildung 49 sind die genannten Kennzeichen dargestellt.

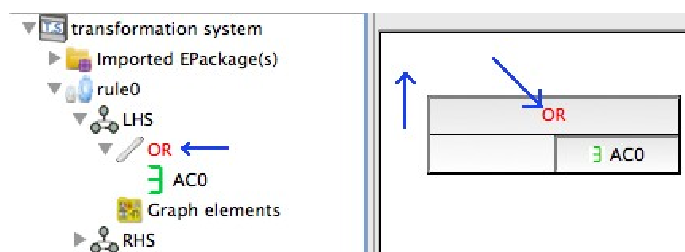


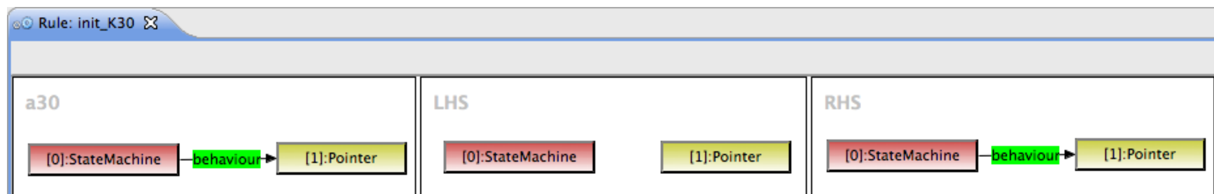
Abbildung 49: Darstellung von einer unvollständigen Anwendungsbedingung

Anforderung 14: ***/Erkennen von PAC und NAC/***

Der ausgebaute Henshin-Editor unterscheidet die Darstellung von *NAC* und *PAC*. In der Baumansicht und in der Blockfigur wird links neben jeder *NAC* das Symbol ∄ angezeigt und links neben jeder *PAC* das Symbol ∃. Dadurch lassen sich *NAC* und *PAC* leicht voneinander unterscheiden.

Anforderung 15: ***/Visualisierung von Mappings zwischen Knoten/***

Die gemappten Knoten werden im Henshin-Editor mit derselben Nummerierung und Farbe versehen. Dadurch sind sie für den Nutzer leicht als solche erkennbar.



Anforderung 16: ***/Darstellung von Amalgamation-Units/***

Amalgamation-Units werden im ausgebauten Henshin-Editor sowohl in der Baumansicht als auch in der Transformation-Unit-Ansicht dargestellt (Abbildung 50).

In der Baumansicht werden Amalgamation-Units als Eintrag unter dem Transformationssystem und dem Container *Transformation Units* repräsentiert. Die Kern- und die Multiregeln werden als innere Einträge einer Amalgamation-Unit dargestellt.

Ein Doppelklick auf eine Amalgamation-Unit in der Baumansicht öffnet die Transformation-Unit-Ansicht, in der die Amalgamation-Unit als Blockfigur dargestellt wird. Die Kern- und die Multiregeln sind in der Figur als innere Blöcke zu sehen.

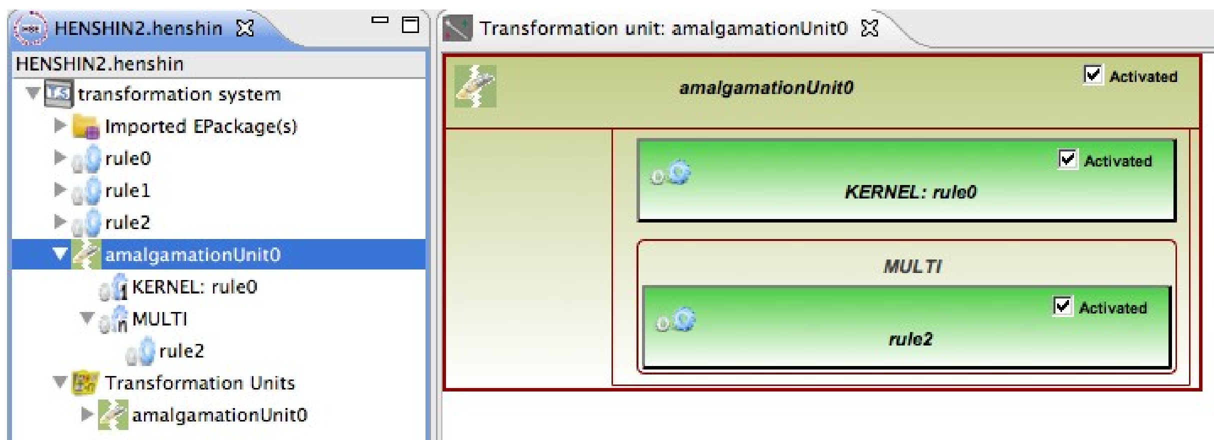


Abbildung 50: Darstellung einer Amalgamation-Unit als Baum und Blockfigur

Anforderung 17: ***/Darstellung von Kernregel und ihre Multiregeln/***

Der ausgebaute Henshin-Editor bietet indirekt eine Übersicht von einer Kernregel und ihren Multiregeln. Auf eine eigene Ansicht für Amalgamation-Units, auf der sowohl die Kernregel als auch ihre Multiregeln gleichzeitig zu sehen sind, wurde verzichtet, weil der Nutzer die geöffneten Kern- und Multiregelansichten so anordnen kann, dass sie gleichzeitig zu sehen sind (z.B. die Kernregelansicht oben und die Multiregelansicht unten, siehe Abbildung 51).

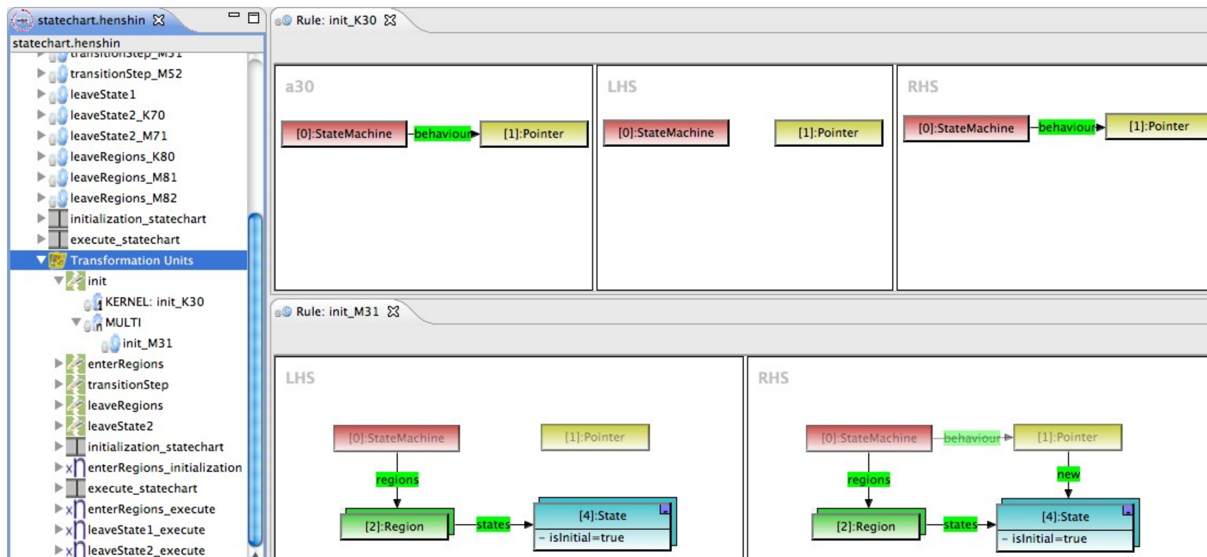


Abbildung 51: Anordnung von Ansichten einer Kern- und einer Multiregel

Anforderung 18: ***/Darstellung von Kern- und Multiknoten/***

Der ausgebaute Henshin-Editor zeigt die Knoten in den Multiregeln mittels zweier Darstellungen an (Abbildung 52).

- Knoten, die aus der Kernregel stammen, werden wie bisher rechteckig dargestellt. Zur Verdeutlichung, dass die Knoten aus der Kernregel stammen und deswegen nicht zu editieren sind, wird das Rechteck etwas heller dargestellt.
- Knoten, die Multiobjekte darstellen, werden durch zwei hintereinander versetzte Rechtecke dargestellt.



Abbildung 52: Darstellung von Knoten in Multiregeln

5.2 Diskussion von gewählten Lösungsansätze im Vergleich mit Alternativen

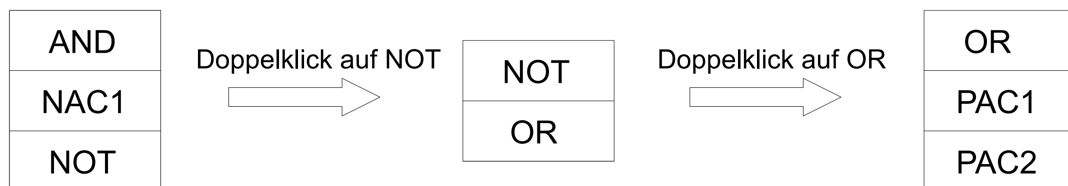
Dieser Abschnitt erörtert alternative Lösungsansätze zu einigen der gestellten Anforderungen.

5.2.1 Darstellung von Anwendungsbedingungen

Die folgende alternative Darstellungsmöglichkeit für Anwendungsbedingungen kam in der Anfangsphase dieser Arbeit auf.

In der Regelansicht sollte nur einen Operator (*NOT*, *AND* oder *OR*) und ihre direkten Operanden angezeigt werden. In dem ersten Kasten (von oben sehen) soll ein Operator dargestellt werden und in den unteren Kästen seine Operanden. Wenn der Operand ein logischer Operator ist, soll die Blockfigur von diesem Operanden durch einen Doppelklick geöffnet werden können. In der neuen Blockfigur soll der Operand als Operator im ersten Kasten zu sehen sein und in den unteren Kästen seine Operanden.

Beispiel: Darstellung von der Anwendungsbedingung *AND (NAC1, NOT (OR (PAC1, PAC2)))*



Der andere Lösungsansatz der Blockfigur wurde ausgewählt, weil sie mehr Übersichtlichkeit von komplexen Anwendungsbedingungen bietet und dort die gesamte Anwendungsbedingung auf einem Blick erkennbar ist.

5.2.2 Visualisierung von Mappings zwischen Knoten

Die Mappings zwischen Knoten wurden am Anfang nur durch gemeinsame Farben visualisiert. Während der Entwicklung des erweiterten Henshin-Editor, hat sich herausgestellt, dass dies allein nicht ausreicht.

In *[BESW10]* wurde der Henshin-Editor mit der erweiterten Funktionalität unter anderem bezüglich der komplexen Anwendungsbedingungen vorgestellt. In schwarz-weiß ausgedruckten Exemplaren waren die Mappingfarben nicht zu erkennen. Aus diesem Grund werden Mappings nicht nur durch gleiche Farben gekennzeichnet sondern zusätzlich durch eine gemeinsame Nummerierung.

5.2.3 Darstellung von Kern- und Multiregeln

In der Anfangsphase war für Amalgamation-Units eine eigene Ansicht geplant (*Amalgamation-Unit View*). Diese Ansicht sollte in zwei Bereiche unterteilt sein. Der obere für die Kernregel und der untere für die Multiregeln. Im unteren Bereich sollten die Multiregeln über zwei Pfeiltasten (nach rechts und links) durchblättert werden können.

Die folgenden Gründe führten dazu, warum diese Lösung nicht umgesetzt wurde:

1. Eine Amalgamation-Unit ist eine Spezialisierung von Transformation-Units. In der Bachelorarbeit von J. Schmidt über Transformation-Units wurde bereits eine Ansicht implementiert, die Transformation-Units darstellt (*TransformationUnit View*). Diese Ansicht öffnet sich per Doppelklick auf eine Transformation-Unit (Sequential-Unit, Conditional-Unit, Priority-Unit, Independent-Unit, Counted-Unit und jetzt auch Amalgamation-Unit) und stellt sie als Blockfigur dar (siehe Abbildung 50 rechts).
   Eine spezielle Ansicht für Amalgamation-Units würde bei gleichen Nutzeraktionen ein unterschiedliches Verhalten der Anwendung zur Folge haben.

2. Der Henshin-Editor verwendet das Framework Muvitor. Mit diesem werden die Inhalte von Views nebeneinander dargestellt. Dies ist in der Regelansicht zu sehen. Auf der linken Seite werden existierende Anwendungsbedingungen dargestellt, in der Mitte *LHS*-Graphen und rechts *RHS*-Graphen. Um die gewünschte Ansicht implementieren zu können, hätte zuerst MuvitorKit erweitert werden müssen. Dies hätte den zeitlichen Rahmen dieser Arbeit gesprengt. Da die Eclipse RCP zudem Nutzern die Möglichkeit bietet, Ansichten selbständig anordnen zu können, wurde auf die Implementierung der *Amalgamation-Unit View* verzichtet. Zudem zeigt eine Multiregelansicht bereits auch die Kernregel mit an.

6 Implementierung

Dieses Kapitel beschreibt zuerst die Frameworks, die für die Implementierung des Henshin-Editors verwendet wurden. Anschließend werden die im Rahmen dieser Arbeit vorgenommenen Erweiterungen am Henshin-Editor erläutert. Hierzu gehören weitere Teile des Henshinmodells, die bisher nicht betrachtet wurden und Erweiterungen am Quelltext.

6.1 Verwendete Frameworks

6.1.1 Eclipse RCP

Der Henshin-Editor wurde in der Programmiersprache Java in der Entwicklungsumgebung Eclipse auf Basis der Eclipse-RCP (Rich Client Plattform) implementiert.

Die Eclipse-RCP ist ein Framework zur Entwicklung graphischer Benutzeroberflächen. Ursprünglich wurde das Framework als Basis für das Eclipse SDK entwickelt, einer Entwicklungsumgebung für Programmierer. Aufgrund der positiven Erfahrungen, insbesondere mit der Plug-in-Struktur auf der alle Eclipse-Anwendungen basieren, wurde die Plattform ab der Version 3.0 zu einer allgemeinen Plattform für Desktop und Rich-Client-Anwendungen ausgebaut.

Die Gestaltung graphischer Benutzeroberflächen basiert auf den Komponenten SWT (Standard Widget Toolkit) und JFace. SWT stellt grundlegende Steuerelemente zur Verfügung. Im Gegensatz zu Swing sind diese nicht vollständig in Java programmiert, sondern verwenden intern die auf dem jeweiligen Betriebssystem verfügbaren Steuerelemente. Dies hat den Vorteil, dass die Steuerelemente nicht auf jeder Plattform gleich aussehen, sondern so, wie der Nutzer der jeweiligen Plattform es erwartet. Dadurch ähnelt die Optik von Programmen, die mit SWT entwickelt wurden, der von nativ entwickelten Programmen.

Da die von SWT zur Verfügung gestellten Komponenten sehr simpel sind, ist deren Verwendung bei der Programmierung relativ aufwendig. JFace löst dieses Problem, indem es aus den primitiven SWT-Elementen komplexere Komponenten zusammenstellt und eine Abstraktionsschicht (Viewer) für den Zugriff auf sie bereitstellt. Dies erleichtert die Programmierung auf Basis von SWT/JFace sehr.

Der Henshin-Editor ist als Eclipse-Plug-in implementiert worden. Dadurch kann der Editor als eigene Eclipse-Anwendung laufen oder aber auch bestehende Eclipse-Installationen erweitern. Im Rahmen dieser Arbeit wurde das bestehende Eclipse-Plugin erweitert.

6.1.2 EMF (Eclipse Modeling Framework)

EMF ist ein Open-Source-Java-Framework für die Modellierung und automatisierte Codegenerierung basierend auf einem strukturierten Datenmodell. *[EMF10, WP-EMF10]*

EMF-Modelle werden in XMI (XML Metadata Interchange) spezifiziert. EMF unterstützt die Entwicklung von Anwendungen durch folgende Eigenschaften:

- Die Modellinstanzen werden in einem Editor in einem Baum dargestellt. Die Modelldefinitionen können hier auch editiert werden.
- Die Quelltexte in der Sprache Java lassen sich aus einem definierten Ecore-Modell mit EMF generieren. Es werden drei Packages generiert, die als Basis für GEF dienen. Die Packages enthalten
  1. die Interfaceklassen für jeden Symboltyp aus dem Ecore-Modell
  2. die Implementierung der Interfaces
  3. zusätzliche Utility-Klassen
- EMF bietet Adapterklassen für kommandobasiertes Editieren von Modellen.

EMF kann Metamodelle sowohl aus Java-Klassen, UML-Diagrammen als auch aus XML-Dateien importieren. *[EBM09]*

EMF-Modelle können auf unterschiedlichen Wegen erstellt werden: *[EBM09]*

- aus annotierten Java-Klassen
- aus XML-Dokumenten
- aus Modellierungstools wie Rational Rose
- direkt mit Hilfe des EMF Ecore Baum-Editors
- mit dem graphischen EMF-Editor-Plug-In von GMF (Graphical Modeling Framework)

Die Modellinstanzen in GMF werden graphisch als UML-Diagramm dargestellt. Die GMF-Dateien haben `.ecore_diagramm` als Dateiendung und die Ecoremodelle `.ecore`. Die beiden Dateien werden beim Speichern automatisch synchronisiert. Daher spielt es kei-

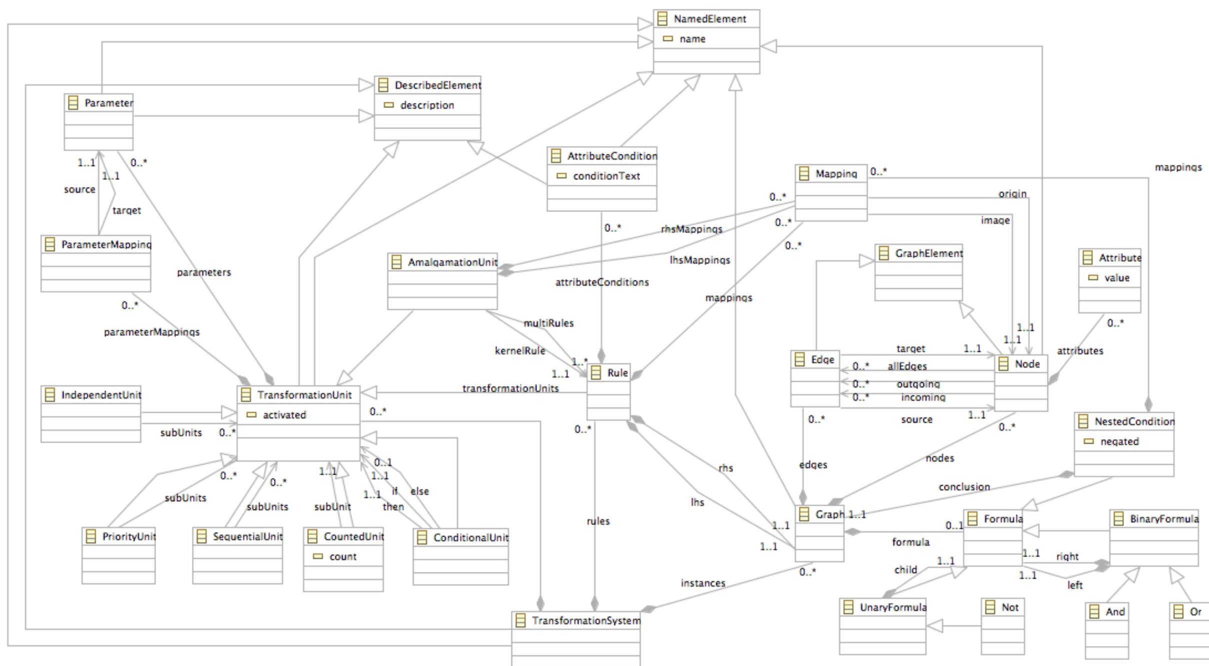


Abbildung 53: Das vollständige Henshin als EMF-Modell

ne Rolle, womit EMF-Modelle definiert sind. EMF und GMF bieten ein frühzeitiges Erkennen von Fehlern in EMF-Modellen durch Auswahl des Menüpunkts *Validate* im Kontextmenü von `.ecore`- oder `.ecore_diagramm`-Dateien.

Die im Henshin-Editor dargestellte Objekte stammen aus dem Henshinmodell, das mit EMF bzw. GMF definiert wurden. Mit EMF wurden die entsprechenden Interfaces, Factory, Utilities und Objekte in Java generiert, die beim Entwickeln des Henshin-Editors eine enorme Rolle spielen.

6.1.3 GEF (Graphical Editing Framework)

GEF ist ein Eclipse-Plug-in, mit dem graphische Editoren für beliebige Datenmodelle (z.B. EMF-Modelle) mit dem Model-View-Controller-Prinzip entwickelt werden können. GEF stellt die Basisklassen zur Implementierung eines graphischen Editor-Plug-ins in Eclipse bereit. Die Schnittstellen für Interaktionen zwischen Nutzern und Modelle, wie

- Verarbeitung von Nutzereingaben per Maus oder Tastatur,
- Veränderung des Modells: Auswählen, Erzeugen und Löschen von graphischen Symbolen,
- Wiederherstellen bzw. Rückgängigmachen von Änderungen (Redo bzw. Undo)

stehen zum Implementieren bereit. GEF bietet ebenfalls nützliche Workbench-Funktionen, wie Aktionen, Menüs, Palette, Toolbars und Keybindings.

Mit GEF lassen sich viele unterschiedliche Arten von Anwendungen erstellen, z.B. GUI-Builder, Diagrammeditoren für UML (z.B.: das GMF-Ecore-Editor-Plug-In) und XML-Editoren.

6.1.3.1 GEF-Architektur

In diesem Abschnitt wird das MVC-Prinzip von GEF kurz erläutert.

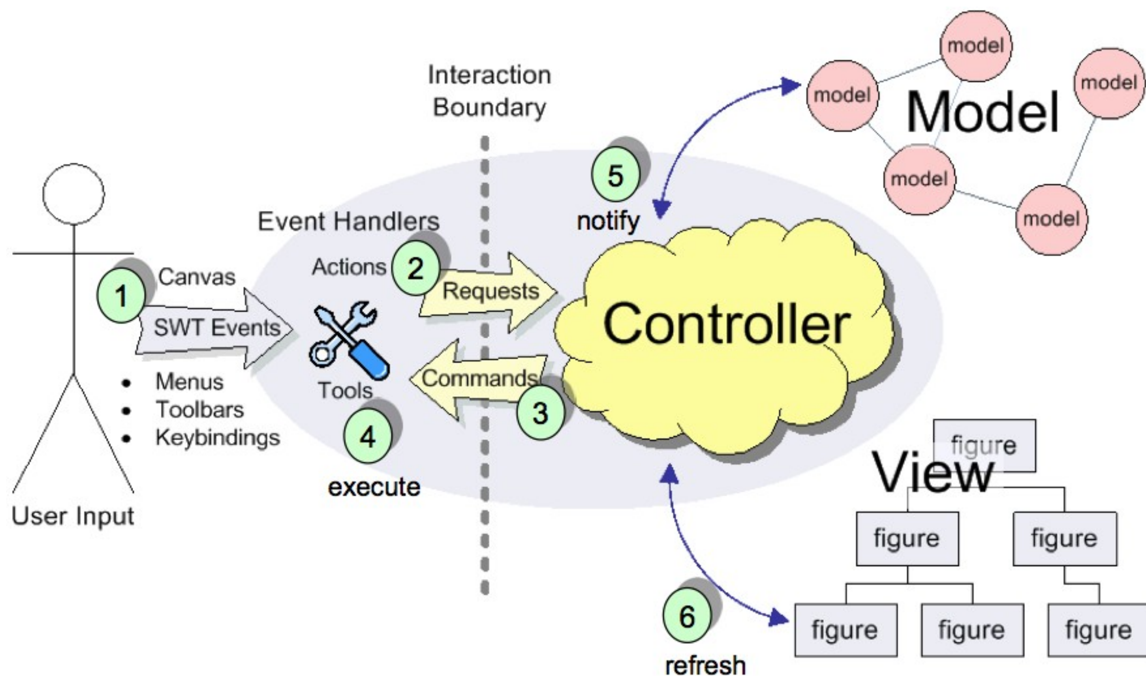


Abbildung 54: GEF-Architektur *[EBM08]*

Durch Nutzereingaben werden Actions ausgelöst und Requests an einen zugehörigen Controller geschickt. Der Controller erzeugt einen passenden Command, der dann ausgeführt wird. Bei einer Modelländerung wird der Controller benachrichtigt, der dann seine Views aktualisiert.

Die GEF-Architektur ist in drei klar voneinander getrennten Schichten gegliedert: *[EBM08]*

- Daten im **Modell**

  Alle persistenten und wichtigen Daten werden ausschließlich hier gespeichert. Jedes in Form von Javaklassen vorliegende Datenmodell wird von GEF akzeptiert (z.B. EMF).

Modelle kennen keine anderen Teile des Programms. Modelländerungen werden über Commands ausgelöst und durch einen Notification-Mechanismus anderen Programmteilen mitgeteilt, die sich als Adapter registriert haben.

- Darstellung in graphischen **Views**

  Views enthalten keine Daten und keine Modelllogik. Sie kennen ebenfalls keine anderen Teile des Programms. Daten der Modellschicht werden als Draw2d-Figures oder als SWT-TreeItems abgebildet.

- Kommunikation zwischen Modell und Views im **Controller**

  Ein Controller stellt die Verbindung zwischen einem Modell und einem View her. In GEF wird diese Aufgabe von EditParts übernommen. EditParts müssen das Interface `org.eclipse.gef.EditPart` implementieren. Sie leiten die Kommunikation vom Modell an den View weiter. Zu jedem EditPart gehören genau ein Modell und ein View.

6.1.3.2 Muvitor

Muvitor steht für Multi-View-Editor und ist ein Eclipse-Plug-in zum Entwicklen von graphischen Editoren. Muvitor ist eine vereinfachte Variante von GEF, in dem einige Standardfunktionalitäten, wie z.B. Copy, Cut, Paste, Revert und ein Notification-Mechanismus bereits implementiert sind.

Muvitor erlaubt das Anzeigen von mehreren Teil-Views in einem Haupt-View. Dies ist in der Regelansicht vom Henshin-Editor zu sehen. Dort werden drei Teil-Views (Anwendungsbedingung, *LHS* und *RHS*) in einem View angezeigt.

6.2 Henshinmodell

Bevor die Implementierung von den im Abschnitt 4.5 gestellten Anforderungen erläutert wird, werden zuerst die Teile vom Henshinmodell vorgestellt, auf denen diese Arbeit basiert. Die in Abbildung 55 und 56 blau markierten Teile, um die es in dieser Arbeit hauptsächlich geht, wurden bei der Entwicklung des Henshin-Editors während des Visuelle-Sprachen-Projekts nicht betrachtet.

6.2.1 EMF-Modell von Anwendungsbedingung

Im Abschnitt 2.3.1 wurde bereits erwähnt, dass ein Graph eine Formel haben kann und eine Formel entweder eine *NestedCondition* ist oder mehrere mit dem logischen Operator *AND* verbundene *NestedCondition* sind. Neu in dem in Abbildung 55 dargestellten Henshin-Modell ist, dass

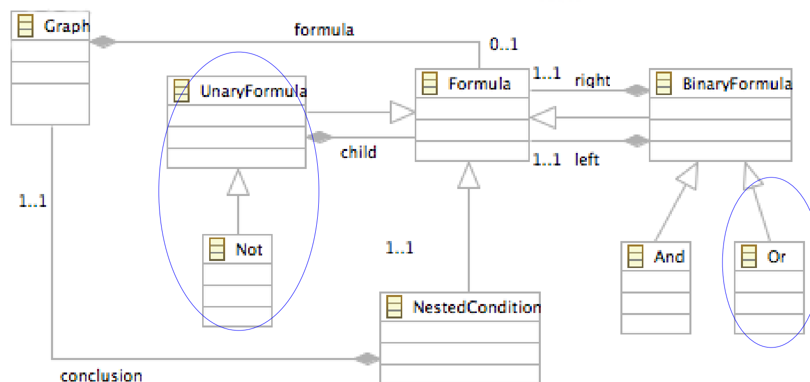


Abbildung 55 : EMF-Modell von komplexen und geschachtelten Anwendungsbedingungen

- nicht nur der logische Operator *AND* eine binäre Formel ist, sondern auch *OR*,
- eine Formel sowohl eine binäre als auch eine unäre Formel sein kann und
- eine unäre Formel ein *NOT* ist.

Die Definition von geschachtelten Anwendungsbedingungen ist schon in dem Henshin-Basismodell enthalten. Sie war nur nicht Teil des Visuelle-Sprachen-Projekts. Daher wurde dieser Teil nicht implementiert. Die Verschachtelung von Anwendungsbedingungen ist in Abbildung 55 dadurch gekennzeichnet, dass eine *NestedCondition* einen Graphen enthalten muss und dieser Graph über eine Formel verfügen kann.

6.2.2 EMF-Modell von Amagamation-Units

In Abbildung 56 sind einige Teile bereits im Abschnitt 2.3.1 vorgestellt worden. Zu den bekannten Teilen sind folgende Definitionen neu hinzugekommen:

- Eine Transformation-Unit kann sowohl eine Regel als auch eine Amalgamation-Unit sein.
- Eine Amalgamation-Unit hat genau eine Kernregel.
- Eine Amalgamation-Unit kann aus mehreren Multiregeln bestehen.

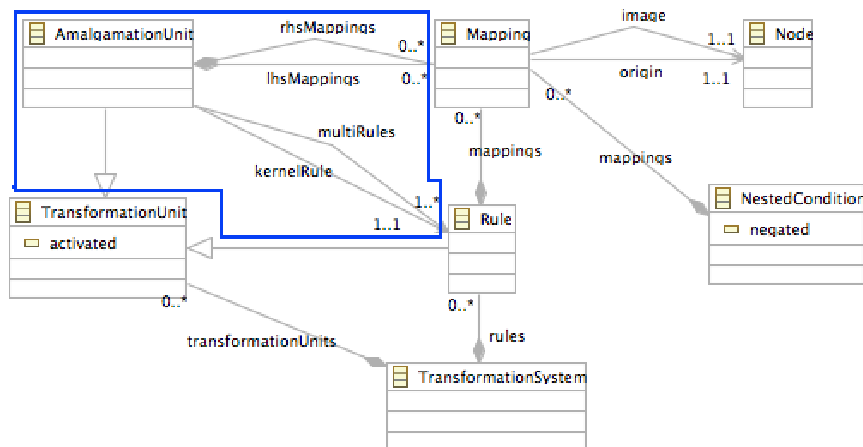
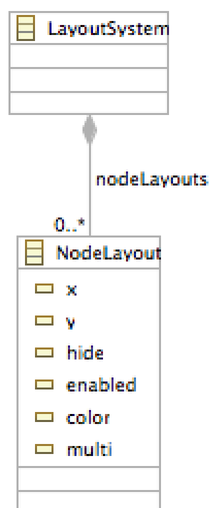


Abbildung 56: EMF-Modell von Amalgamation-Units

- Amalgamation-Units speichern die Informationen über die Mappings zwischen den Knoten eines *LHS*- bzw. *RHS*-Graphen aus der Kernregel und den Knoten eines *LHS*- bzw. *RHS*-Graphen aus der jeweiligen Multiregel in *lhsMappings* bzw. *rhsMappings*. Diese Informationen sind wichtig, um Kernknoten in Multiregeln grafisch abbilden zu können.

6.3 Henshinlayoutmodell



Das Henshinlayoutmodell wurde im Abschnitt 2.3.2 bereits vorgestellt. Um den Henshin-Editor ausbauen zu können, sind folgende neue Eigenschaften für *NodeLayout* notwendig:

- Mit *enabled* kann die Darstellung eines Knotens in der Multiregelansicht bestimmt werden. Diese Eigenschaft hat den Typ *boolean*. Der Wert dieser Eigenschaft ist nur *false*, wenn der darzustellende Knoten ein Kernknoten in einer Multiregel ist.
- Mit *color* kann die Mappingfarbe eines Knotens gespeichert und in Knoten in der Regelansicht angezeigt werden.
- Mit *multi* kann die Darstellung eines Knotens in der Multiregelansicht bestimmt werden. Besitzt diese Eigenschaft den Wert *true*, hat der Knoten eine Multiknoten-Darstellung (mit zwei hintereinander versetzte Vierecke). Andernfalls wird er in einem einfachen Rechteck dargestellt.

6.4 Implementierung der Anforderungen

In diesem Abschnitt werden die Implementierungen zu den gestellten Anforderungen bezüglich der Anwendungsbedingungen und Amalgamation-Units erläutert. Im Allgemeinen gibt es drei Bereiche, die implementiert werden müssen:

- Actions

  Die Kontextmenüs selektierbarer Objekte wurden um zusätzliche Menüpunkte erweitert.
  Jede Aktion wird in einer eigenen Klasse implementiert, die die Klasse `org.eclipse.gef.ui.actions.SelectionAction` erweitern. Das heißt, die Aktionen sind im Henshin-Editor erst sichtbar, wenn etwas selektiert ist. Einige Methode müssen bzw. können überschrieben werden.

  - `calculateEnabled()` zum Steuern der Sichtbarkeit von Aktionen bei selektierten Objekten
  - `getImageDescriptor()` zum Anzeigen von Icons links neben der Aktionsbeschriftung
  - `run()` zum Ausführen eines passenden Commands von einer ausgewählten Aktion.

  Die neuen Aktionen werden im Package `henshineditor` in der Klasse `HenshinTreeEditor` in der Methode `createCustomActions()` registriert und in der Klasse `HenshinTreeContextMenuProvider` werden die Kontextmenüs mit der Aktionenauswahl als Menü- bzw. Untermenüpunkte in der Methode `buildContextMenu(IMenuManager menu)` erzeugt.

- Commands

  Commands führen die vom Nutzer ausgewählten Aktionen aus und verändern intern das Modell bzw. das aktuelle Objekt, an dem die Aktion ausgeführt werden soll. Jede durch Commands ausgeführte Änderung kann rückgängig gemacht und wiederhergestellt werden. Die Command-Klassen im Henshin-Editor erweitern entweder `org.eclipse.gef.commands.Command` für ein einzelnes Command oder

`org.eclipse.gef.commands.CompoundCommand`, wenn sie mehrere Commands beinhalten. Es sind drei wichtige Methoden in Command-Klassen zu implementieren:

- `execute()` zum Ausführen des Commands
- `undo()` zum Rückgängigmachen einer Aktion nach `execute()`
- `redo()` zum Wiederholen einer Aktion nach `undo()`

Um zu bestimmen, ob ein Command ausführbar bzw. rückgängig zu machen ist, kann die Methode `canExecute()` bzw. `canUndo()` überschrieben werden.

- EditParts

Wie schon weiter oben beschrieben wurde, übernehmen EditParts die Rolle eines Controllers. Sie sind jeweils als Adapter einer Modellklasse registriert und werden deshalb durch den Notification-Mechanismus über Änderungen im Modell benachrichtigt. Um auf Modelländerungen zu reagieren, muss die Methode `notifyChanged()` implementiert werden.

Im Henshin-Editor werden Objekte sowohl in einer Baumansicht als auch in einer graphischen Ansicht dargestellt. EditPart-Klassen, die Objekte in einer Baumansicht darstellen, erweitern die Klasse `muvitorkit.gef.editparts.AdapterTreeEditPart`. Solche die Objekte in einer graphischen Ansicht darstellen, erweitern die Klasse `muvitorkit.gef.editparts.AdapterGraphicalEditPart`. In EditPart-Klassen sind drei wichtige Methoden zu Implementieren:

- `createFigure()` bzw. `setText()` zum Erstellen von Figuren in der graphischen Ansicht bzw. des Textes in der Baumansicht,
- `refreshVisuals()` zum Aktualisieren der Daten der Viewschicht mit den Daten der Modellschicht und
- `getModelChildren()` zum Bestimmen einer Liste von Modellklassen, die logische Kinder von dem zum EditPart korrespondierenden Modellelement sind.

In EditParts sind Editierrollen (EditPolicies) zum Behandeln spezieller Anfragen (Requests) zu installieren. Hierfür dient die Methode `createEditPolicies()`. Ein EditPart iteriert über alle seine EditPolicies, um Requests zu bearbeiten. Mit den Informationen eines Request wird ein passendes Command erzeugt.

In EditParts kann bestimmt werden, ob die Eigenschaften von dem aktuell anzuzeigenden Modell in der Properties-Ansicht angezeigt werden sollen. Hierfür muss die Methode `createPropertySource()` überschrieben und die entsprechende Property-Klasse zurückgeliefert werden.

`org.eclipse.gef.EditPartFactory` ist ein Interface, das dafür sorgt, dass richtige EditParts erzeugt werden. Die Klasse hat genau eine Methode zu implementieren: `EditPart createEditPart(EditPart context, Object model);`
Anhand des gegebenen Kontexts und des aktuellen Modells kann bestimmt werden, welches konkrete EditPart zu erzeugen ist. Die Klasse `HenshinTreeEditPartFactory` im Package `henshineditor.editparts.tree` implementiert dieses Interface und erstellt die TreeEditParts im Henshin-Editor.

6.4.1 Implementierung des Editors für Anwendungsbedingungen

6.4.1.1 Actions

Im Package `henshineditor.actions.condition` befinden sich die implementierten Klassen zu den neuen Aktionen, die mit dem Erstellen, Editieren und Löschen von Anwendungsbedingungen zu tun haben. Die Aktionen werden als Menüpunkte zum Auswählen im Kontextmenü eines bestimmten Objektes in der Baumansicht zur Verfügung gestellt.

- `CreateConditionAction` implementiert den Menüpunkt *Create Condition Tree...*. im Kontextmenü eines *LHS*- und *AC*-Graphen. Dieser Menüpunkt ist nur sichtbar, wenn der selektierte Graph noch keine Anwendungsbedingung hat.
- `CreateFormulaAction` ist die Oberklasse von der Action-Klassen `CreateApplicationConditionAction`, `CreateNotAction`, `CreateAndAction` und `CreateOrAction`, die jeweils die Menüpunkte *Create Application-Condition*, *Create Not-Condition*, *Create And-Condition* und *Create Or-Condition* implementieren. Diese Menüpunkte werden im Kontextmenü eines *LHS*-Graphen, eines *AC*-Graphen und eines logischen Operators (*NOT*, *AND*, *OR*) angezeigt und sind nur dann sichtbar, wenn
  - der selektierte Graph noch keine Anwendungsbedingung hat oder
  - die Operanden von dem selektierten Operator noch nicht vollständig sind.

Der Menüpunkt *Create Application-Condition* stellt eine Ausnahme dar. Auch wenn der selektierte Graph bereits über eine Anwendungsbedingung verfügt, wird er sichtbar. Dies ermöglicht es dem Nutzer, einem Graphen immer eine Application-Condition hinzuzufügen.

- `SetNegatedAction` implementiert die Menüpunkte *Set negated = true* und *Set negated = false* im Kontextmenü eines *PAC* bzw. *NAC*, von denen immer nur einer sichtbar ist.
- `SwapBinaryFormulaAction` implementiert den Menüpunkt *Swap AND* → *OR* bzw. *Swap OR* → *AND* im Kontextmenü eines *AND* bzw. *OR*.

6.4.1.2 Commands

Das Package `henshineditor.commands.condition` enthält die Command-Klassen zum Erstellen, Editieren und Löschen von Anwendungsbedingungen.

- `CreateConditionCommand` fügt eine Formel in eine Prämisse ein. Hier wird dafür gesorgt, dass eine Formel vom Typ *ApplicationCondition* eine Konklusion mit einem generierten eindeutigen Namen hat. Hat die Prämisse noch keine Formel, wird die neue einfach eingefügt. Andernfalls wird zuerst eine *AND*-Formel erzeugt und danach werden die alte und die neue Formel als Operanden dieses *AND* eingefügt.
- `CreateFormulaCommand` fügt eine Formel in eine Parentformel ein. Hier wird dafür gesorgt, dass eine Formel vom Typ *ApplicationCondition* eine Konklusion mit einem eindeutigen (generierten) Namen hat.
- `SetNegatedCommand` ändert den *negated*-Wert einer *NAC* zu *false* und den *negated*-Wert einer *PAC* zu *true*.
- `SwapBinaryFormulaCommand` tauscht die binären Formeln *AND* und *OR*. Abhängig von der im Konstruktor angegebenen binären Formel wird die andere binäre Formel erzeugt und die Operanden der alten Formel in die neue Formel eingefügt. Dabei wird dafür gesorgt, dass das Elternobjekt, das die zu tauschende Formel enthält, ebenfalls aktualisiert wird. Dies bedeutet, dass einem Graphen eine neue Formel zugewiesen wird und einem *NOT*, *AND* oder *OR* ein neuer Operand.

6.4.1.3 EditParts

Das Package `henshineditor.editparts.condition` beinhaltet die Ableitungen der Klasse EditPart zum Darstellen von Anwendungsbedingungen in der graphischen Ansicht.

- `ConditionEditPart` beinhaltet ein Layer zum Anzeigen von Anwendungsbedingungen in Form einer Blockfigur. Hier wird die aktuelle Formel auch in einer Textdarstellung angezeigt.
- `ApplicationConditionEditPart`, `UnaryFormulaEditPart` und `BinaryFormulaEditPart` präsentieren eine Formel (*AC*, *NOT*, *AND*, *OR*) als Blockfigur.
  Die Formelfiguren werden im Package `henshineditor.figure.condition` implementiert. Jede Figur hat einen Text. Figuren von *Application Condition* enthalten zusätzlich ein Icon links neben dem Text, um *NAC* von *PAC* unterscheiden zu können.

Im Package `henshineditor.editparts.tree.condition` befinden sich die von EditPart abgeleiteten Klassen zur Darstellung der Anwendungsbedingungen in der Baumansicht. Eine *Application Condition* wird anders dargestellt als logische Operatoren.

- `ApplicationConditionTreeEditPart` präsentiert nicht die *AC* selbst, sondern ihre Konklusion (den *AC*-Graph). Links neben dem Namen vom *AC*-Graphen wird ein Icon angezeigt, um *NAC* von *PAC* zu unterscheiden.
- `ConditionTreeEditPart` präsentiert die logischen Operatoren. Der anzuzeigende Operatorname ist festgelegt (NOT, AND oder OR) und nicht editierbar.

Im Package `henshineditor.model.properties.condition` befindet sich die Klasse `ApplicationConditionPropertySource` die für die Darstellung der Eigenschaften der *Application Condition* zuständig ist. Der Name und der *negated*-Wert werden in der Properties-Ansicht angezeigt und sind dort auch editierbar.

6.4.2 Implementierung des Editors für Amalgamation-Units

6.4.2.1 Actions

Im Package `henshineditor.actions.transformation_unit` ist eine neue Klasse `CreateAmalgamationUnitAction` hinzugekommen. Sie implemetiert den Menüpunkt *Create amalgamation unit*, der nur sichtbar wird, wenn das Transformationssystem oder der Container *Transformation Units* selektiert ist.

Aufgrund der verschiedenen Arten von Knoten (Kern- und Multiknoten) und Regeln (Kern- und Multiregel) müssen in den existierenden Action-Klassen einige Erweiterungen vorgenommen werden. Dies betrifft folgende Packages:

- `henshineditor.actions.graph`
    - `CreateNodeAction` implementiert den Menüpunkt *Create Node* im Kontextmenü eines Graphen in der Baumansicht. Die Klasse wurde dahingehend erweitert, dass ein passendes Command anhand des selektierten Graphen ausgeführt wird. Falls ein Knoten in einer Kernregel erzeugt werden soll, muss auch je ein Knoten in ihren jeweiligen Multiregeln erzeugt werden.
- `henshineditor.actions.rule`
    - `CreateRuleAction` wurde dahingehend erweitert, dass sie nicht nur den Menüpunkt *Create Rule* implementiert, sondern auch *Create Kernel Rule* und *Create Multi Rule*. Anhand des selektierten Eintrags in der Baumansicht wird die jeweils passende Beschriftung angezeigt. Fallunterscheidungen müssen eingebaut werden, um ein passendes Command auszuführen: Entweder soll eine Regel nur in das Transformationssystem eingefügt werden, oder zusätzlich als Kern- oder Multiregel einer selektierten Amalgamation-Unit definiert werden.
    - `DeleteMappingAction` implementiert den Menüpunkt *Delete Mapping* im Kontextmenü einer Knotenfigur in der graphischen Ansicht. Die Klasse wird dadurch erweitert, dass die Commands zum Löschen von Mappings in Multiregeln auch dann ausgeführt werden, wenn der selektierte Knoten zu einer Kernregel gehört.
    - `LhsToRhsCopyAction` implementiert die Aktion des Buttons LR in der Werkzeugleiste einer Regelansicht. Es wurden folgende Fallunterscheidungen eingebaut:

- Wenn die selektierte Regel weder als Kern- noch als Multiregel definiert ist, werden alle Elemente von *LHS* nach *RHS* kopiert.
- Wenn eine Kernregel selektiert ist, werden alle Elemente von *LHS* einer Kernregel und ihre Multiregeln nach *RHS* kopiert.
- Wenn eine Multiregeln selektiert ist, werden nur Multiobjekte von *LHS* nach *RHS* kopiert.

6.4.2.2 Commands

Die Command-Klassen, die für die Modelländerungen bezüglich Amalgamation-Units sorgen, befinden sich in den Packages

- `henshineditor.commands.graph`
- `henshineditor.commands.rule`
- `henshineditor.commands.transformation_unit`
- `henshineditor.commands.transformation_unit.parameter`

Folgende Command-Klassen wurden neu implementiert und sorgen dafür, dass Modelländerungen sowohl in der aktuellen Kernregel als auch in ihren jeweiligen Multiregeln ausgeführt werden:

- `CreateKernelEdgeCommand` fügt eine Kante zwischen Knoten hinzu.
- `CreateKernelNodeCommand` fügt einen Knoten hinzu. Zu jedem in Multiregeln einzufügenden Knoten wird ein Mapping in *lhsMappings* bzw. *rhsMappings* erstellt, damit die Information über das Abbild der Kernknoten in der entsprechenden Amalgamation-Unit nicht verloren geht.
- `CreateKernelNodeMappingCommand` erzeugt Mappings zwischen Knoten.
- `CreateKernelParameterCommand` erzeugt einen Parameter.
- `KernelMultiRhsCopyCommand` kopiert *LHS*-Elemente nach *RHS*.
- `DeleteNodeCommand` löscht einen Knoten.
- `DeleteEdgeCommand` löscht eine Kante.

Folgende Command-Klassen sind neu hinzugekommen. Sie sorgen dafür, dass Modelländerungen in Multiregeln durchgeführt werden:

- `CreateMultiRuleCommand` erzeugt eine Multiregel mit kopierten *LHS*- und *RHS*-Elementen einer Kernregel und kopierten Parametern.
- `DeleteMultiRuleCommand` entfernt eine Multiregel von einer Amalgamation-Unit.
- `MultiObjectsCopyCommand` kopiert Multiknoten und die dazugehörigen Kanten einer Multiregel von *LHS* nach *RHS*.
- `CopyMapping2MultiRuleCommand` erzeugt Mappings in einer Multiregel anhand gemappter Knoten in der Kernregel.

Folgende Command-Klassen wurden angepasst, um die restlichen gestellten Anforderungen bezüglich Amalgamation-Units zu erfüllen:

- `CreateRuleCommand` wurde um einen Konstruktor erweitert, so dass zusätzlich eine Amalgamation-Unit und eine boolesche Variabel `isKernelRule` als Parameter übergeben werden können. Die neuen Parameter entscheiden, ob eine Regel zusätzlich als Kern- bzw. Multiregel einer Amalgamation-Unit definiert wird.
- `CreateNodeMappingCommand` wurde so erweitert, dass nicht nur Mappings erzeugt werden, sondern auch die Mappingfarbe gesetzt wird.
- `DeleteMappingCommand` wurde dahingehend erweitert, dass nun berücksichtigt wird, ob das zu löschende Mapping aus *lhsMappings* bzw. *rhsMappings* stammt und löscht dies entsprechend mit.
- `DeleteRuleCommand` wurde derartig erweitert, dass eine Regel auch aus der Multiregelliste einer Amalgamation-Unit entfernt wird, bevor sie aus dem Transformationssystem gelöscht wird.
- `DeleteRuleNodeCommand` wurde dergestalt erweitert, dass überprüft wird, ob der zu löschende Knoten aus einer Amalgamation-Unit stammt. In diesem Fall werden die Mappings sowohl aus der Regel als auch aus *lhsMappings* bzw. *rhsmappings* gelöscht.
- `GraphCopyCommand` wurde derart erweitert, dass der Fall, dass ein Graph nicht nur von *LHS* nach *RHS* kopiert werden soll, sondern auch von *LHS* einer Kernregel nach *LHS* ihrer jeweiligen Multiregeln bzw. von *RHS* einer Kernregel nach *RHS* ihrer jeweiligen Multiregeln, berücksichtigt wird.

- DeleteTransformationUnit wurde dahingehend erweitert, dass eine Regel, wenn sie vorher als Kern- oder Multiregel definiert war, nicht sofort aus dem Transformationssystem gelöscht, sondern nur aus der Amalgamation-Unit entfernt wird.

6.4.2.3 EditParts

Im Package henshineditor.editparts.transformation_unit befindet sich die Klasse AmalgamationUnitEditPart zum Anzeigen einer Amalgamation-Unit und ihrer Kern- und Multiregeln in einer Blockfigur in der graphischen Ansicht.

Im Package henshineditor.editparts.tree.transformation_unit befinden sich die das Interface EditPart implementierenden Klassen zum Darstellen von Amalgamation-Units mitsamt ihrer Kern- und Multiregeln in der Baumansicht.

6.4.3 Weitere Hilfsklassen für den Henshin-Editor

Im Package henshineditor wurde die Henshinperspektive um die Bedingungsansicht (*Condition View*) und die Transformation-Unit-Ansicht (*TransformationUnit View*) erweitert.

Alle im Henshin-Editor verwendeten Icons befinden sich im Package henshineditor.icons.

Das Package henshineditor.interne_eobjects enthält selbst implementierte Klassen, die nicht im Henshinmodell definiert sind. Es wurde um die Klasse AmalgamationUnitPart erweitert.

Das Package henshineditor.ui.condition beinhaltet Implementierungen zum *Condition View* mitsamt ihrer Palette und des Kontextmenüs.

Im Package henshineditor.ui.dialog ist die Klasse CreateAmalgamationUnitDialog hinzugekommen, die den Dialog zum Erstellen von Amalgamation-Units implementiert.

Das Package henshineditor.ui.dialog.condition beinhaltet Klassen, die den Dialog zum Erstellen von vollständigen Anwendungsbedingungen implementieren.

Das Package henshineditor.util enthält allgemeine Hilfsklassen, die für die Entwicklung des Henshin-Editors benötigt wurden.

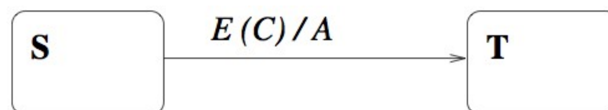
7 Anwendungsbeispiel

In diesem Kapitel wird als Anwendungsbeispiel des Henshin-Editors die operationale Semantik von Statecharts mit amalgamierter Graphtransformation *[GBEE10]* modelliert. Das Kapitel beginnt mit einer Definition und einer Modellierung eines Statecharts. Anschließend wird das Statechartsbeispiel ProdCons in konkreter und abstrakter Syntax vorgestellt. Danach wird der Statechart-Interpreter im Henshin-Editor modelliert. Abschließend kommt eine ausführliche Beschreibung der ersten Ereignisdurchführung und eine tabellarische Darstellung von allen durchzuführenden Ereignissen und die entsprechenden aktuellen Zustände vor und nach der Durchführung.

7.1 Definition

Ein Statechart (Zustandsübergangdiagramm) ist in der Informatik eine Darstellungsform, um den Kontrollfluss von Programmen in Form von Zuständen und Zustandsübergängen wiederzugeben. *[WP-ZUS10]*

Die Notation von Statecharts nach [Har87]:



- Die Kästen präsentieren die Zustände und die Pfeile die Zuständsübergänge.
- E := Ereignis (*engl. Event*); C := Bedingung (*engl. Condition*); A := Aktion (*engl. Action*)

David Harel führte weitere wichtige Notationselemente ein, die für dieses Anwendungsbeispiel relevant sind.

- Hierarchische Komposition ermöglicht die Definition von Unterzuständen. Das heißt, dass ein Zustand einen anderen Zustand beinhalten kann.
- Parallele Komposition ermöglicht ein paralleles Schalten mehrerer Zustände.
- Default-Zustände
  Bei der Aktivierung eines hierarchischen Zustandes muss einer der Unterzustände aktiviert werden.

7.1.1 Modellierung von Statecharts

Bevor das Statechart-Beispiel im Henshin-Editor simuliert wird, muss zuerst das Modell in EMF definiert werden (Abbildung 57).

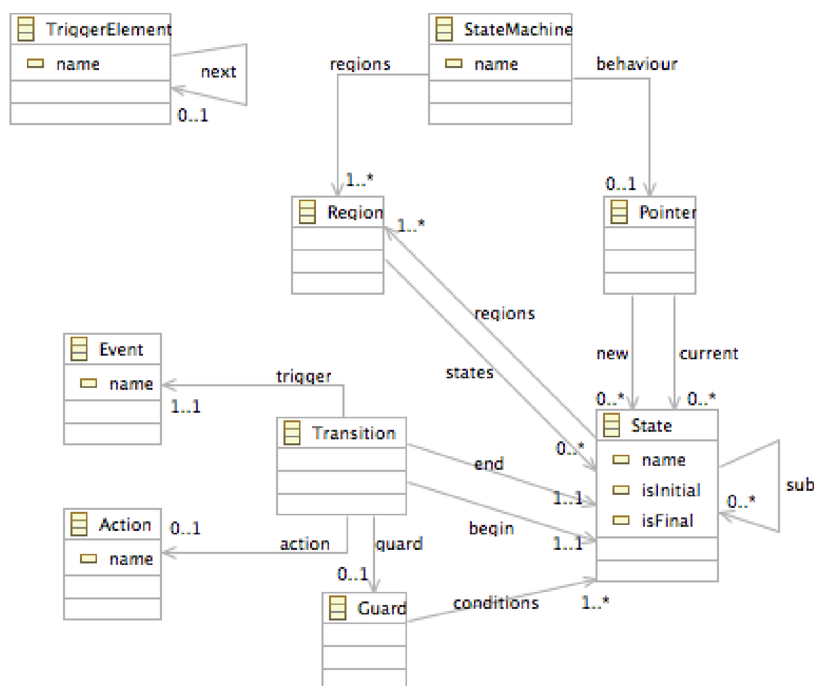


Abbildung 57: EMF-Modell von Statecharts

Einige Integritätsbedingungen können im Statechart-Modell direkt mit Hilfe von Multiplizitäten festgelegt werden. Andere müssen dagegen explizit textuell definiert werden. Im Folgenden werden das Statechart-Modell und seine Integritätsbedingungen genauer erläutert:

- Ein Triggerelement *(TriggerElement)* hat höchstens ein Triggerelement als Nachfolger. Triggerelemente beschreiben Ereignisse, die von der Zustandmaschine behandelt werden müssen.
- Ein Statechart-Instanzgraph muss mindestens ein leeres Triggerelement mit dem Attribut *name = „null“*, einen Zeiger (*Pointer*) und genau eine Zustandsmaschine (*StateMachine*) beinhalten.
- Eine Zustandsmaschine muss mindestens eine Region (*Region*) und darf höchstens einen Zeiger (*Pointer*) haben.
- Ein Zeiger kann auf neue (*new*) oder aktuelle (*current*) Zustände zeigen. Er stellt aktive Zustände in der Zustandsmaschine dar.

- Ein Zustand (*State*) kann wiederum Zustände enthalten. Ein Zustand enthält mindestens eine Region und er muss initialisiert sein. Dies bedeutet, dass entweder das Attribut *isInitial* oder *isFinal* den Wert *true* hat.
- Eine Region kann über mehrere Zustände verfügen und gehört entweder zu einer Zustandsmaschine oder zu einem Zustand.
- Die Namen von Zuständen innerhalb einer Region müssen eindeutig sein.
- Jede Region hat genau einen Initialzustand und höchstens einen Finalzustand. Ein Finalzustand darf über keine Region verfügen.
- Eine Transition (*Transition*) hat genau einen Start- und einen Endzustand. Sie wird von genau einem Ereignis (*Event*) ausgelöst. Außerdem kann sie einen Wächter (*Guard*) haben, der mindestens einen Zustand als Bedingung hat. Eine Transition kann höchsten eine Aktion (*Action*) ausführen.
- Es existiert ein besonderes Ereignis mit dem Attribut *name = „exit“* zum Verlassen eines Zustandes nach dem Beenden aller orthogonalen Regionen. Ein solches Ereignis darf keine Wächterbedingung haben.
- Finalzustände dürfen keiner Transition als Startzustand zugewiesen werden und müssen den Namen *name = „final“* tragen.
- Transitionen dürfen keine Start- und Endzustände zugewiesen bekommen, die aus verschiedenen Regionen mit dem gleichen Oberzustand stammen.

7.2 Statechart-Beispiel „ProdCons“

In dem Statechart-Beispiel „ProdCons“, das in diesem Kapitel simuliert wird, handelt es sich um ein Produzent-Verbraucher-System (*engl. producer-consumer*). Abbildung 58 zeigt das System in konkreter Syntax.

Der Zustand *prod* ist ein Initialzustand und hat drei Regionen, die parallel arbeiten: Produzent, Puffer und Verbraucher.

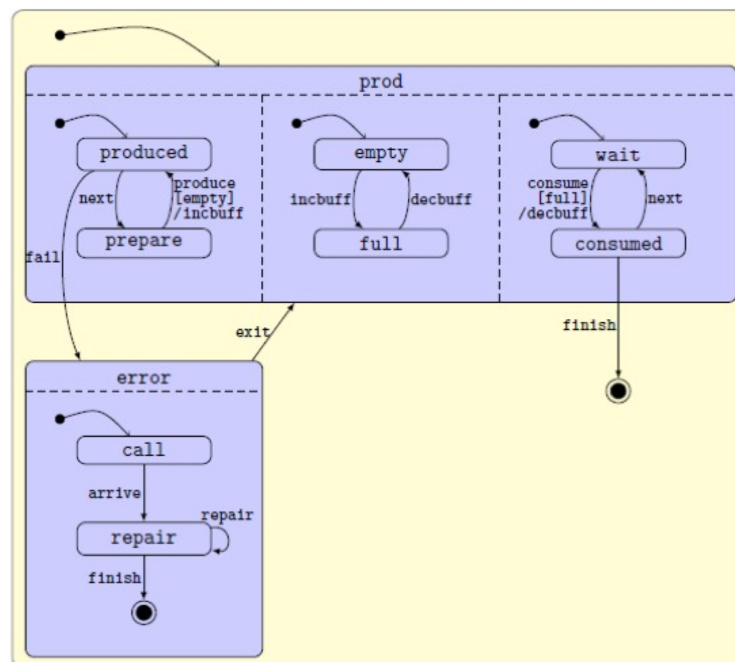


Abbildung 58: Statechart „ProdCons“ in konkreter Syntax *[BEEG]*

- Der Produzent wechselt zwischen den Zuständen *produced* und *prepare*. Der Zustandsübergang *produce* modelliert die aktuelle Produktionsaktivität. Er wird erst durchgeführt, wenn *empty* der aktuelle Zustand in der parallelen Region ist, d.h. der Puffer ist leer und er kann ein Produkt erhalten. Dies ist durch die Aktion *incbuff* modelliert.
- Der Puffer wechselt zwischen den Zuständen *empty* und *full*.
- Der Verbraucher wechselt zwischen den Zuständen *wait* und *consumed*. Der Zustandsübergang *consume* wird erst von dem Zustand *full* bewacht. Dies bedeutet, dass er erst durchgeüfhrt wird, wenn der Puffer voll ist. Danach wird die Aktion *decbuff* ausgeführt, um den Puffer zu entleeren.

Zwei Ereignisse können zum Verlassen des Zustandes *prod* führen:

1. Der Verbraucher hat sich entschieden, den Systemablauf zu beenden.
2. Ein Fehler wurde entdeckt, der nach der Produktion zum *error*-Zustand führt. In diesem Fall muss die Maschine repariert werden, bevor der Zustandsübergang *exit* durchgeführt wird, um die Standardprozesse im Zustand *prod* wieder in Betrieb nehmen zu können.

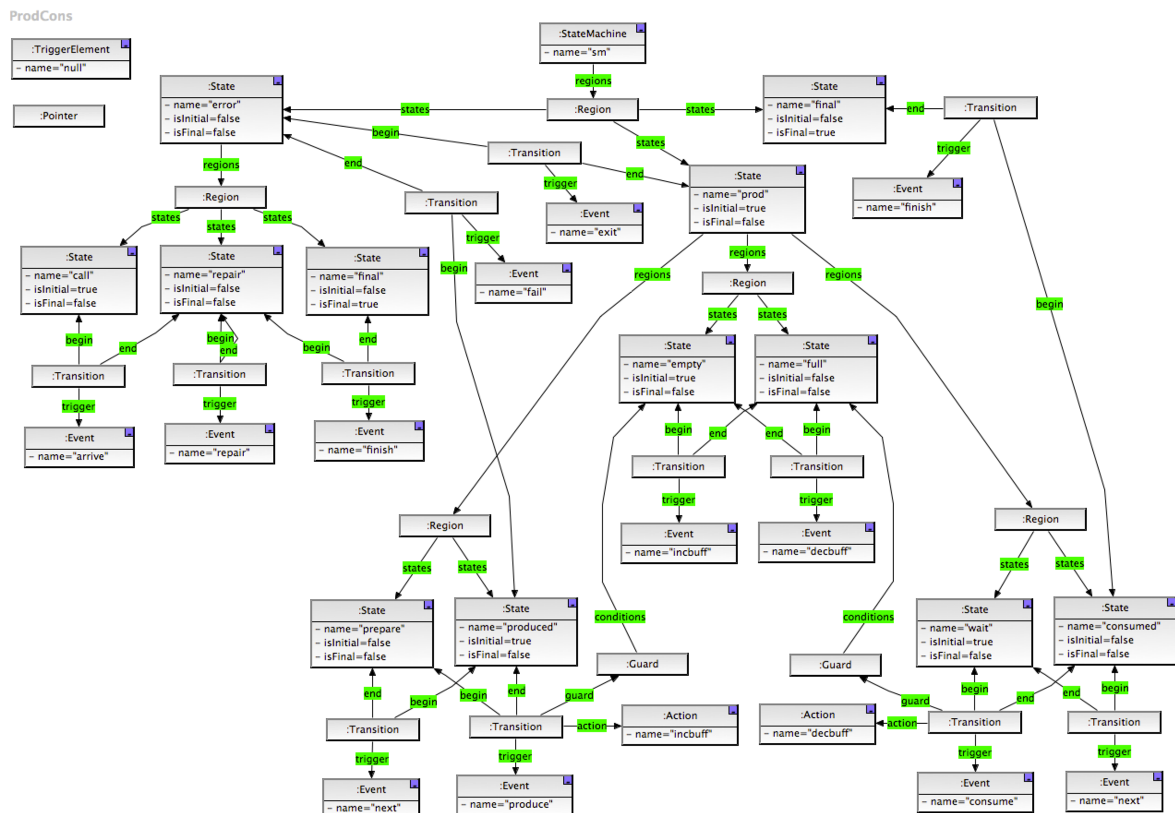


Abbildung 59: ProdCons-Graph im Henshin-Editor

In Abbildung 59 wird das Statechart ProdCons in abstrakter Syntax in Form eines Graphen im Henshin-Editor modelliert. Die Knoten *TriggerElement* und *Pointer* müssen existieren, damit das Statechart-Modell gültig ist. Um eine bessere Übersichtlichkeit zu erschaffen, werden die *sub*-, *new*- und *current*-Kanten hier noch nicht dargestellt.

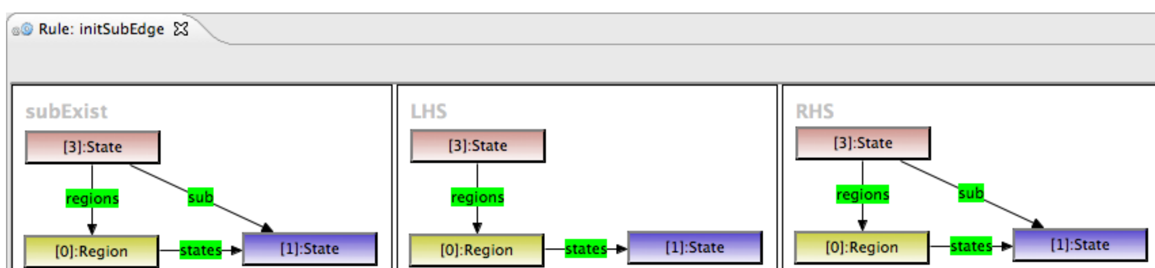
7.3 Statechart-Interpreter

Der Statechart-Interpreter wird in diesem Abschnitt im Henshin-Editor modelliert. Hierfür müssen zuerst Amalgamation-Units mit komplexen Anwendungsbedingungen definiert werden. Die Definition der Interpreter-Semantik wird in zwei Schritte unterteilt: der Initialisierungsschritt und der semantische Schritt.

7.3.1 Initialisierungsschritt

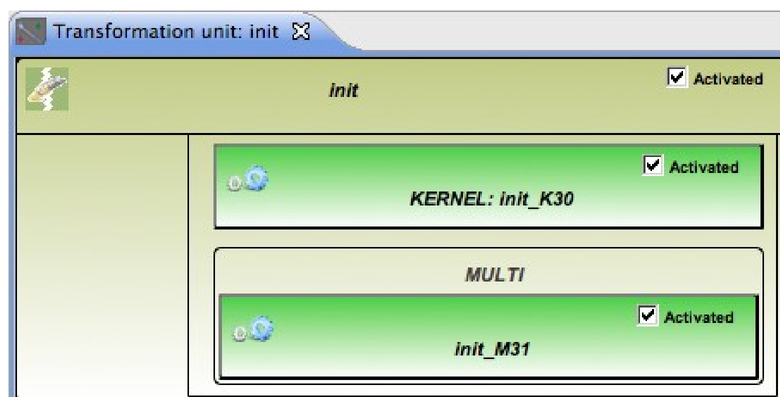
Für den Initialisierungsschritt ist es notwendig, die Reihenfolge der auszulösenden Ereignisse in Triggerelementen zu definieren und die *sub*-Kanten zwischen einem Zustand und seinen Unterzuständen zu erzeugen. Danach kann die Amalgamation-Unit *init* ausgeführt werden, gefolgt mit mehrmaliger Ausführung von *enterRegions*.

Die Regel *initSubEdge*



Diese Regel fügt die Verbindungen zwischen einem Zustand und einem seiner Unterzustände ein.

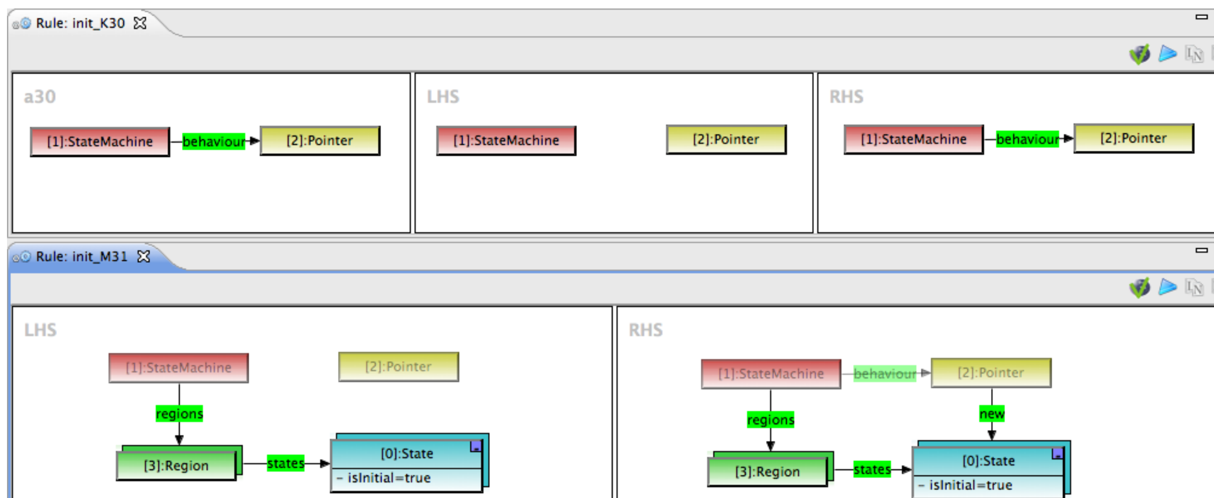
Die Amalgamation-Unit *init*



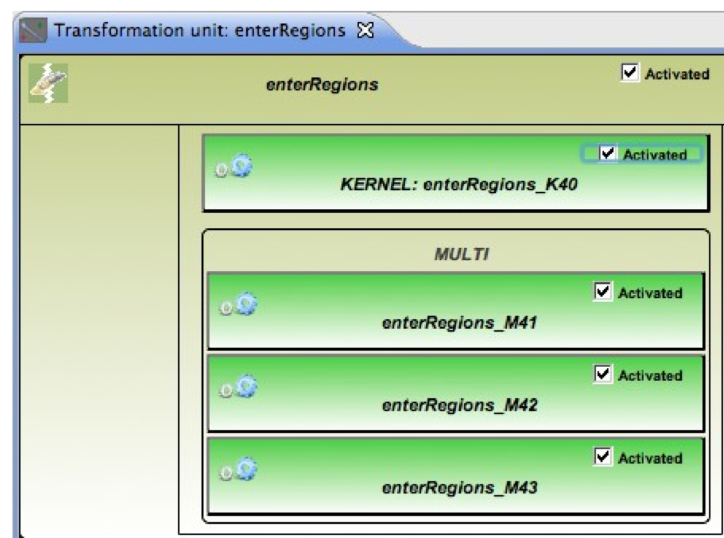
Diese Unit überprüft, ob eine Beziehung zwischen der Zustandsmaschine und dem Zeiger existiert. Ist dies nicht der Fall, werden sie miteinander verbunden. Alle Initialzustände, die zu der Region der Zustandsmaschine gehören, werden mit der new-Kante verbunden. Das heißt, dass

- eine *new*-Kante nicht eingefügt wird, wenn zwischen den Zuständen bereits eine solche oder eine *current*-Kante existiert, und

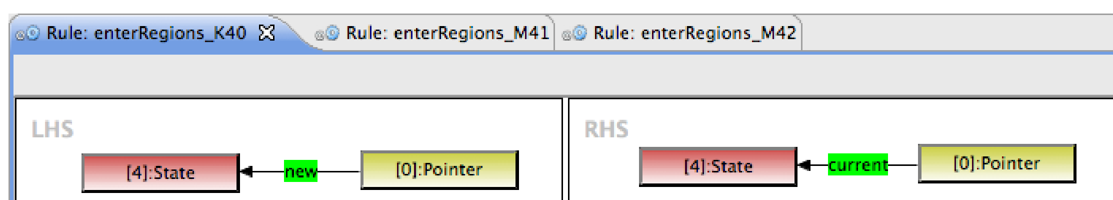
- eine *new*-Kante gelöscht wird, wenn zwischen den Zuständen bereits eine *current*-Kante existiert.



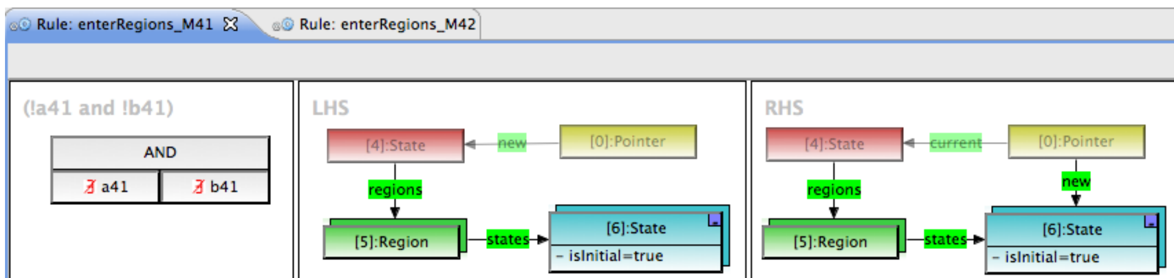
Amalgamation-Unit *enterRegions*



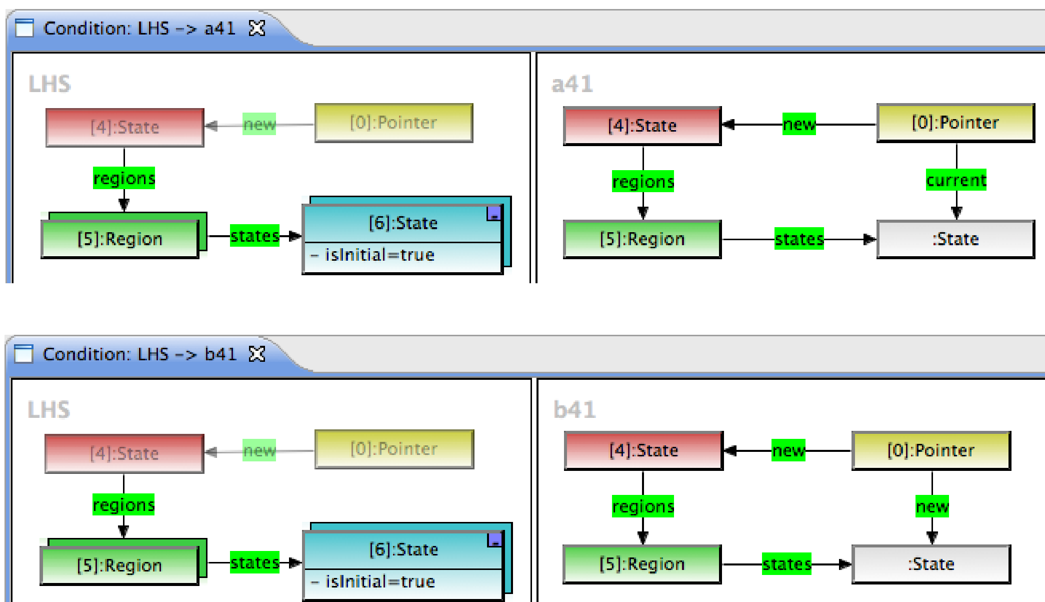
In der Kernregel *enterRegions_K40* wird eine *new*-Kante zwischen einem Zustand und dem Pointer durch *current*-Kante ersetzt.



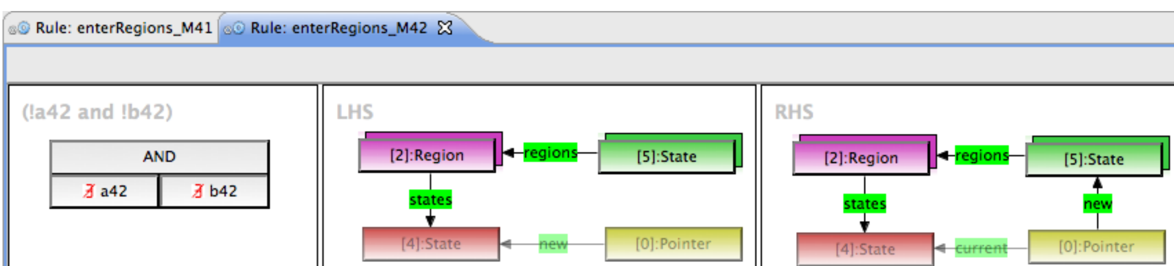
Die Multiregel *enterRegions_M41* fügt eine *new*-Kante zwischen dem *Pointer* und den aktiven Initial-Unterzuständen des in der Kernregel genannten Zustandes hinzu.



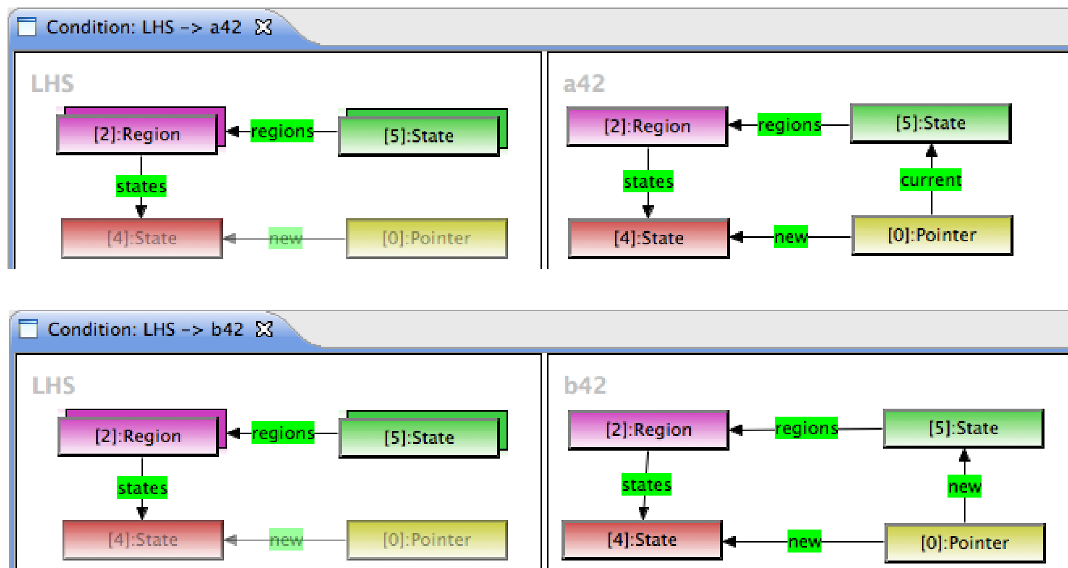
Die Regel wird erst ausgeführt, wenn die Anwendungsbedingung $LHS \rightarrow \nexists a41 \quad \nexists b41$ erfüllt ist. Die Bedingung stellt sicher, dass nur initiale Zustände aktiviert werden. Dies bedeutet, dass vor der Regelanwendung keine aktiven oder aktuellen Unterzustände in der Region existieren dürfen.



Die Multiregel *enterRegions_M42* fügt eine *new*-Kante zwischen dem *Pointer* und Oberzuständen des in der Kernregel genannten Zustandes hinzu.



Die Regel wird erst ausgeführt, wenn keine Verbindung zwischen dem Pointer und den Oberzuständen existiert: $LHS \rightarrow \nexists a42 \quad \nexists b42$.

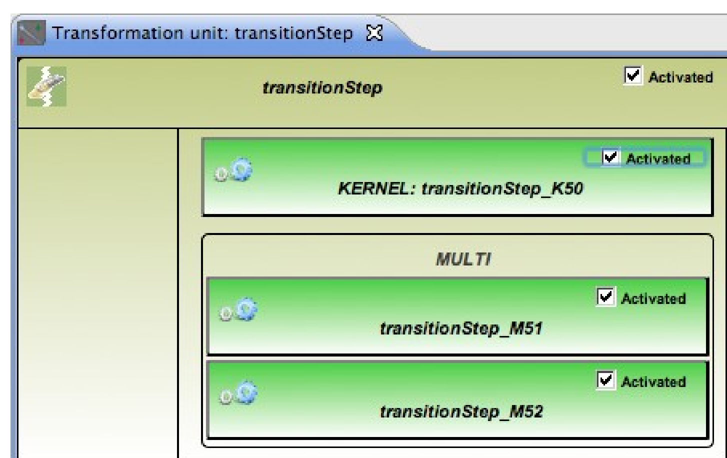


Die Amalgamation-Unit *enterRegions* beinhaltet auch die mit der Kernregel identischen Multiregel *enterRegions_M40*. Diese Regel stellt sicher, dass ein Match von einer der Multiregeln im Arbeitsgraphen gefunden wird, um die Unit auszuführen.

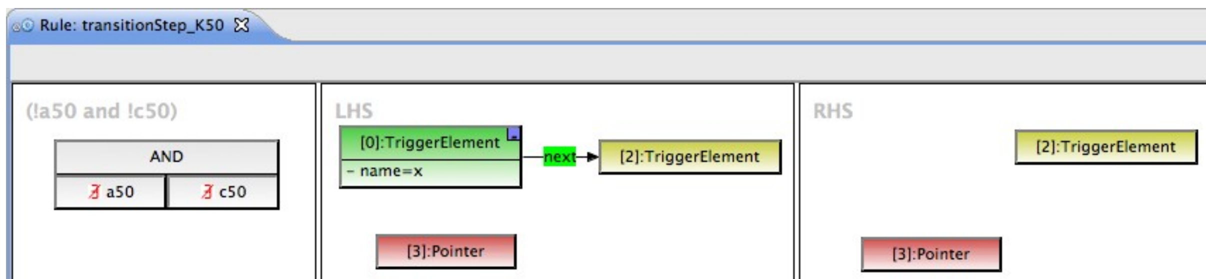
7.3.2 Semantischer Schritt

Das Wechseln von einem Zustand zu einem anderen erfolgt durch Ausführung der Amalgamation-Unit *transitionStep*, gefolgt mit mehrmaliger Ausführung der Amalgamation-Units *enterRegions*, *leaveState1*, *leaveState2* und *leaveRegions*.

Amalgamation-Unit transtitionStep

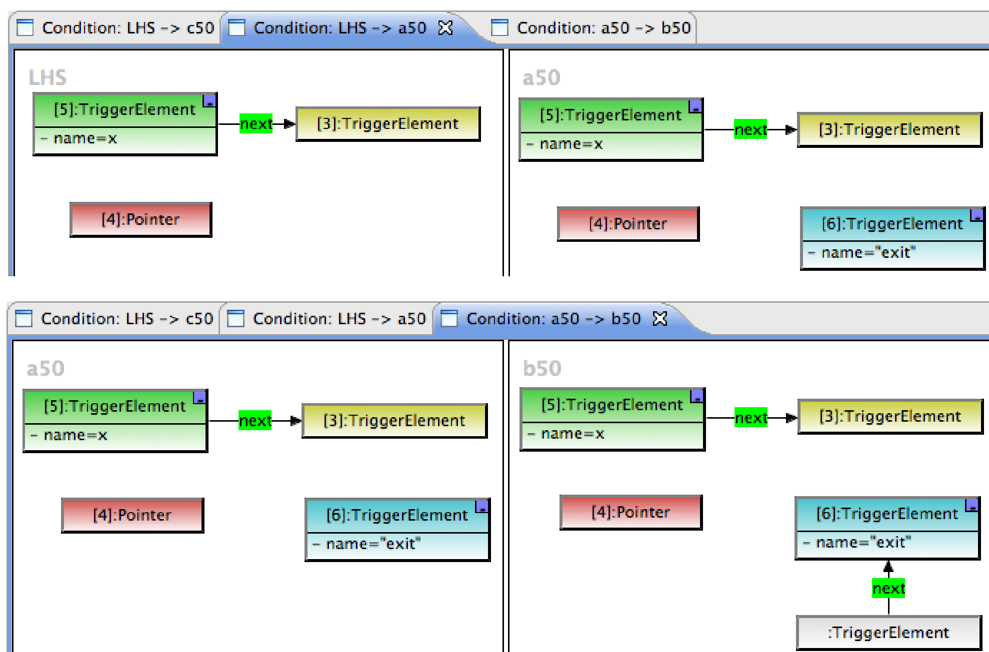


In der Kernregel *transitionStep_K50* wird das erste Triggerelement ausgewählt und gelöscht.

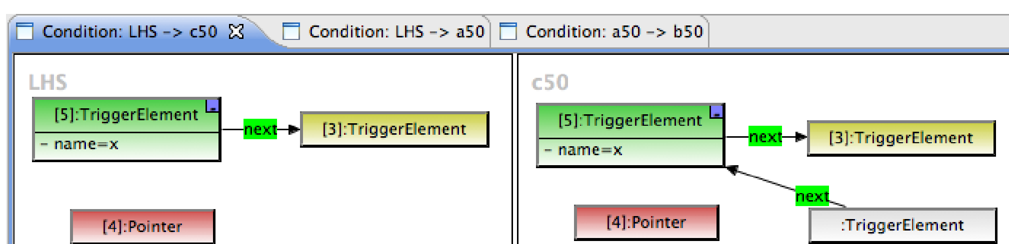


Die Kriterien zum Auswählen vom Triggerelementen sind in der Anwendungsbedingung der Kernregel festgelegt: $LHS \rightarrow (\nexists a50 \rightarrow \nexists b50) \quad \nexists c50$

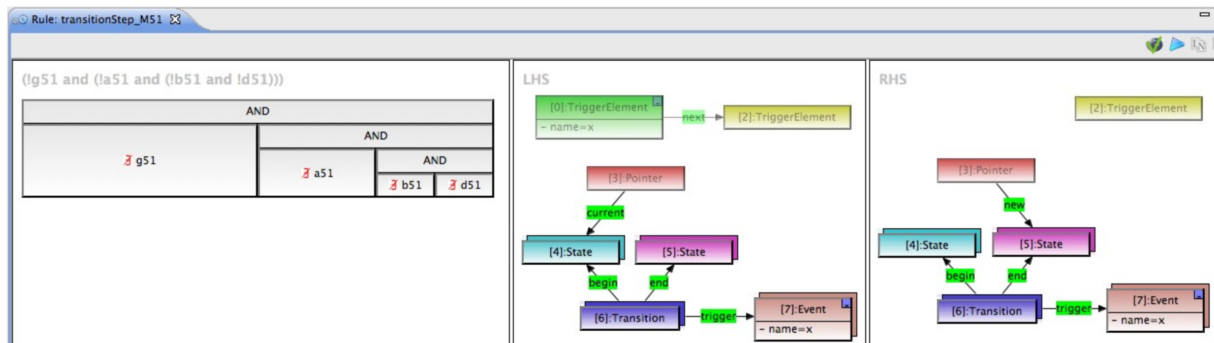
1. Ein Triggerelement mit *name = „exit"* muss vorrangig behandelt werden.



2. Das auszuwählende Triggerelement darf nicht mit zwei anderen Triggerelementen verbunden sein.



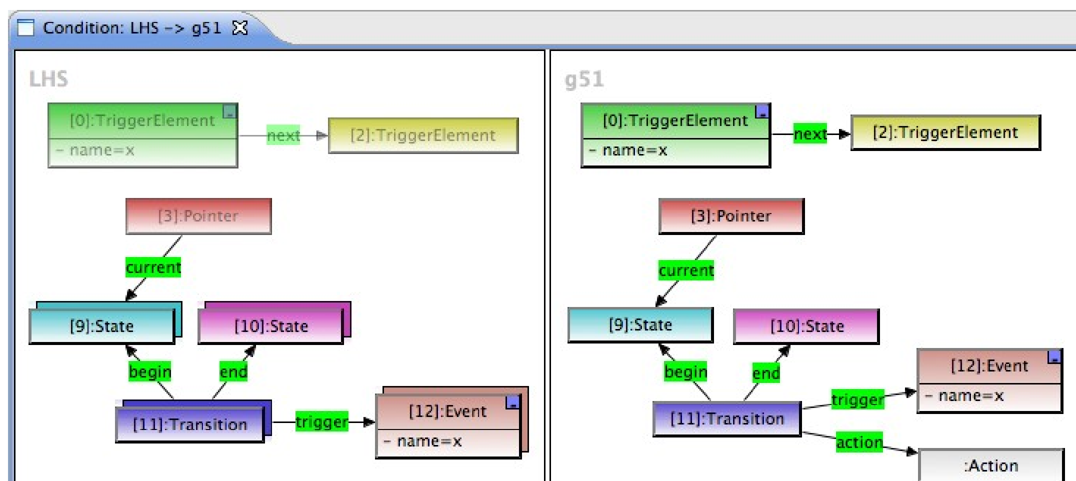
Die Multiregel *transitionStep_M51* führt ein Ereignis mit dem gleichen Namen wie der des gelöschten Triggerelementes durch. Nach der Regelanwendung wird der Zeiger mit der *new*-Kante nicht mehr auf den Startzustand zeigen, sondern auf den Endzustand.



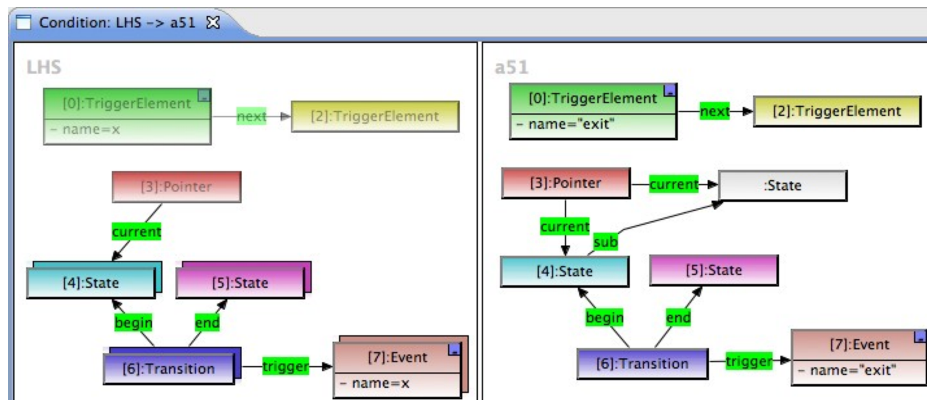
Die Transition mit dem durchzuführenden Ereignis ist erst dann aktiv, wenn die vier mit dem logischen UND verbundenen Anwendungsbedingungen erfüllt sind:

$LHS \rightarrow \nexists g51 \quad \nexists a51 \quad (\nexists b51 \rightarrow \nexists c51) \quad (\nexists d51 \rightarrow (\nexists e51 \rightarrow \nexists f51))$

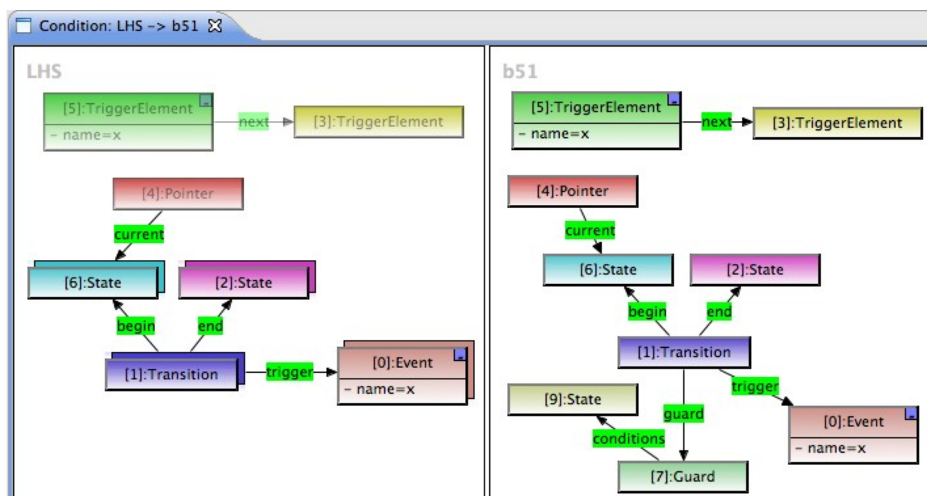
1. Der Zustandübergang darf keine Aktion auslösen.

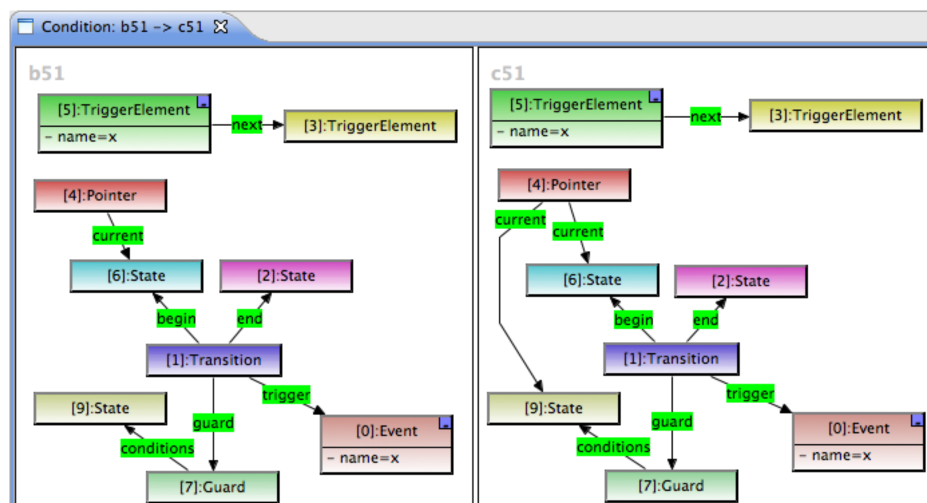


2. Der Startzustand, von dem die Transition ausgehen soll, muss aktiv sein.

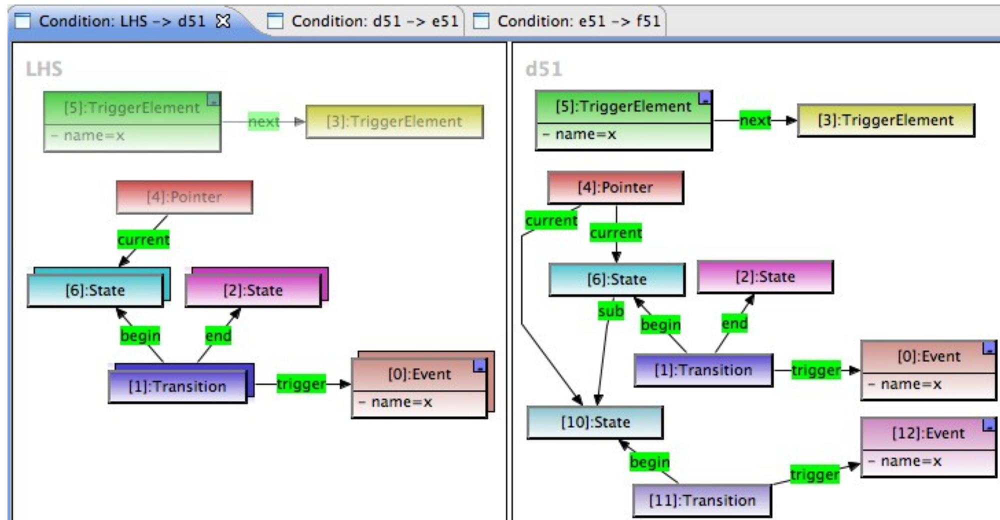


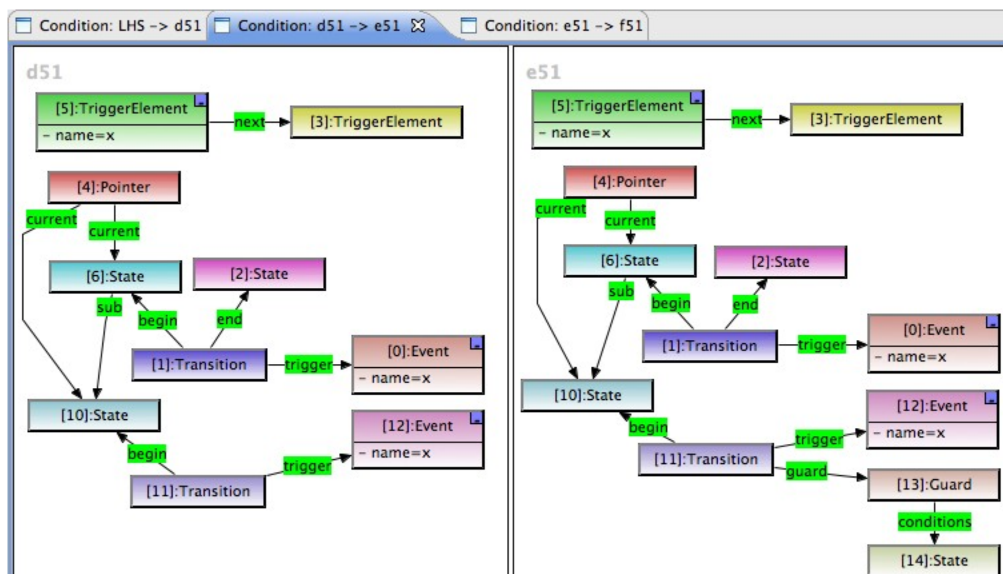
3. Der Bedingungszustand, der die Transition überwacht, muss aktiv sein.

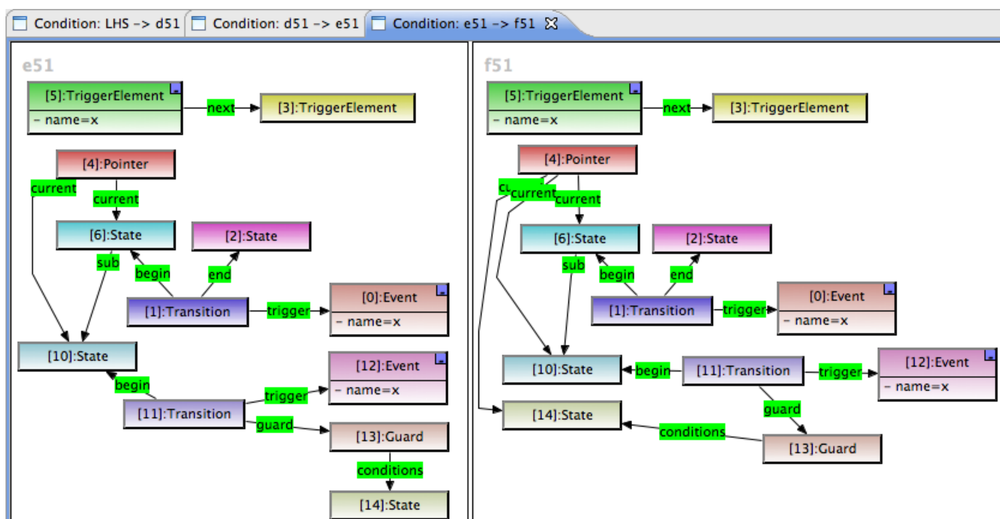




4. Die Transition darf keine aktiven Unterzustände haben, die mit einer anderen aktiven Transition mit dem gleichen Namen wie das durchzuführende Ereignis als Startzustand in Verbindung stehen. Für die Unterzustände gelten wieder die Bedingungen 2 und 3.


Condition: LHS -> d51
Condition: d51 -> e51
Condition: e51 -> f51
LHS
[5]:TriggerElement
- name=x
next
[3]:TriggerElement
[4]:Pointer
current
[6]:State
[2]:State
begin
end
[1]:Transition
trigger
[0]:Event
d51
sub
[10]:State
[11]:Transition
[12]:Event

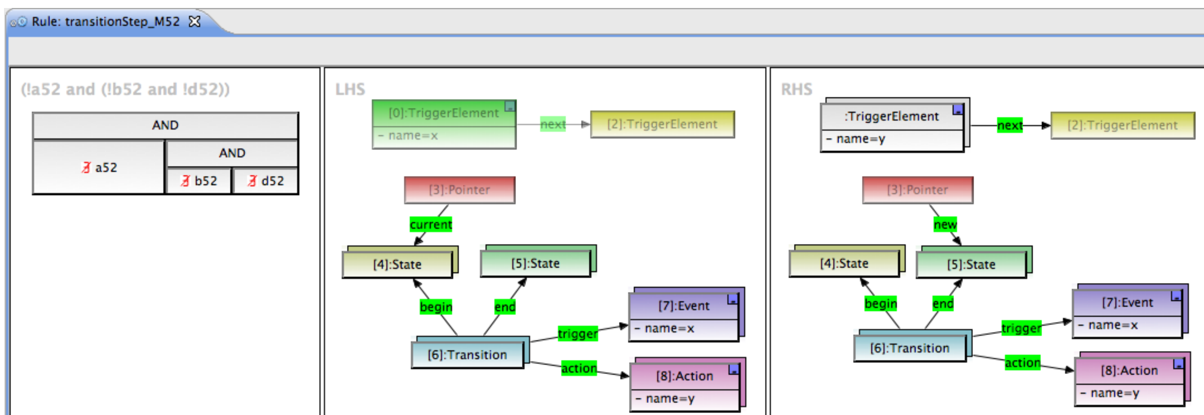

Condition: LHS -> d51
Condition: d51 -> e51
Condition: e51 -> f51
d51
e51
[5]:TriggerElement
- name=x
next
[3]:TriggerElement
[4]:Pointer
current
[6]:State
[2]:State
sub
begin
end
[1]:Transition
trigger
[0]:Event
[10]:State
[11]:Transition
[12]:Event
guard
[13]:Guard
conditions
[14]:State


Condition: LHS -> d51
Condition: d51 -> e51
Condition: e51 -> f51
e51
f51
[5]:TriggerElement
- name=x
next
[3]:TriggerElement
[4]:Pointer
current
[6]:State
[2]:State
sub
begin
end
[1]:Transition
trigger
[0]:Event
[10]:State
[11]:Transition
[12]:Event
guard
[13]:Guard
conditions
[14]:State

Die Multiregel *transitionStep_M52* verfährt ähnlich wie *transitionStep_M51*. Der Unterschied liegt darin, dass hier Transitionen behandelt werden, die eine Aktion auslösen. Daher hat diese Regel nur drei miteinander verbundene Anwendungsbedingungen zu erfüllen:

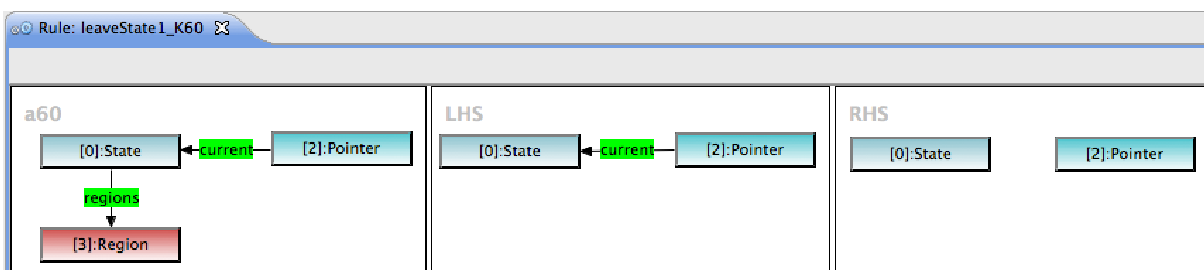
$$LHS \rightarrow \nexists a52 \quad (\nexists b52 \rightarrow \nexists c52) \quad (\nexists d52 \rightarrow (\nexists e52 \rightarrow \nexists f52))$$

Die 1., 2. und 3. Operanden entsprechen den 2., 3. und 4. Operanden der Anwendungsbedingung von *transitionStep_M51*. Die *AC*-Graphen in *transitionStep_M52* enthalten jeweils einen zusätzlichen *Action*-Knoten (siehe die unten dargestelle Regel *transitionStep_M52*).

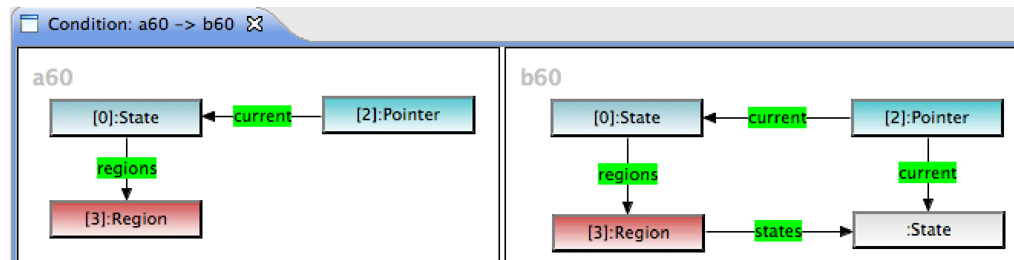


leaveState1, *leaveState2* und *leaveRegions* sorgen dafür, dass aktive Zustände korrekt ausgewählt werden.

Amalgamation-Unit *leaveState1*

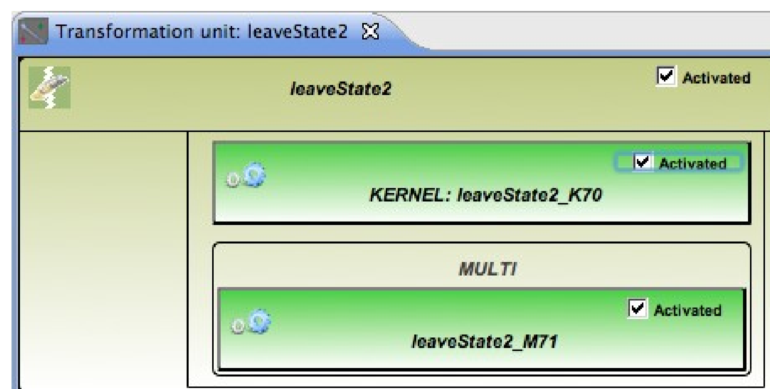


Diese Unit löscht die *current*-Kante von einem Zustand mit nicht aktiven Unterzuständen. Dies bedeutet, dass es keine aktiven Oberzustände geben darf, wenn alle Unterzustände in einer ihrer Regionen nicht aktiv sind. Die Überprüfung ist in der Anwendungsbedingung abgebildet: $LHS \rightarrow (\exists a60 \rightarrow \nexists b60)$.

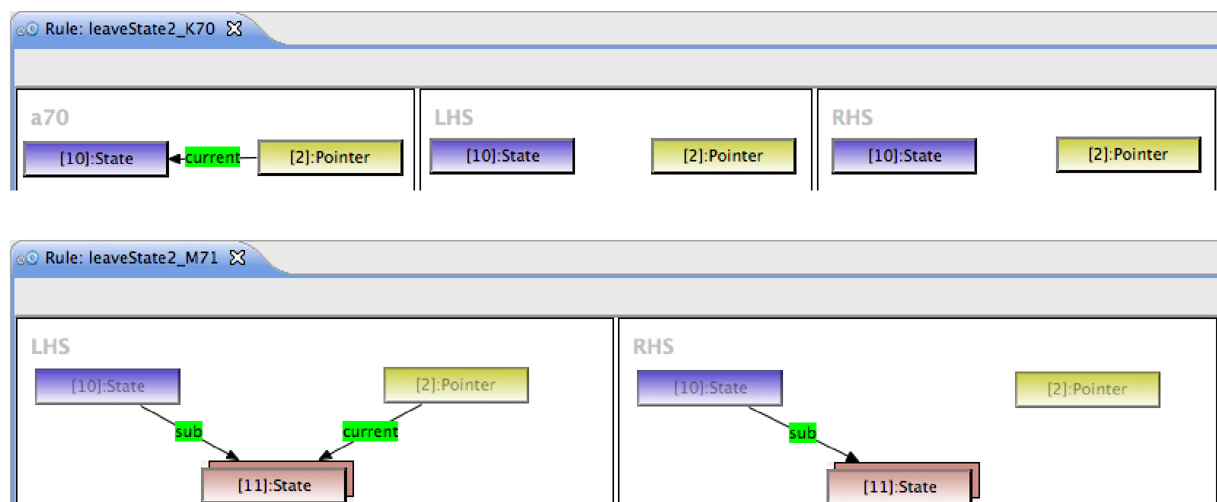


Die Multiregel *leaveState1_M60* ist identisch mit der Kernregel. Sie hat keine Anwendungsbedingung. Das führt dazu, dass die Amalgamation-Unit ausgeführt wird, wenn ein Match von dem *LHS*-Graphen der Kernregel im Arbeitsgraphen vorgefunden werden kann.

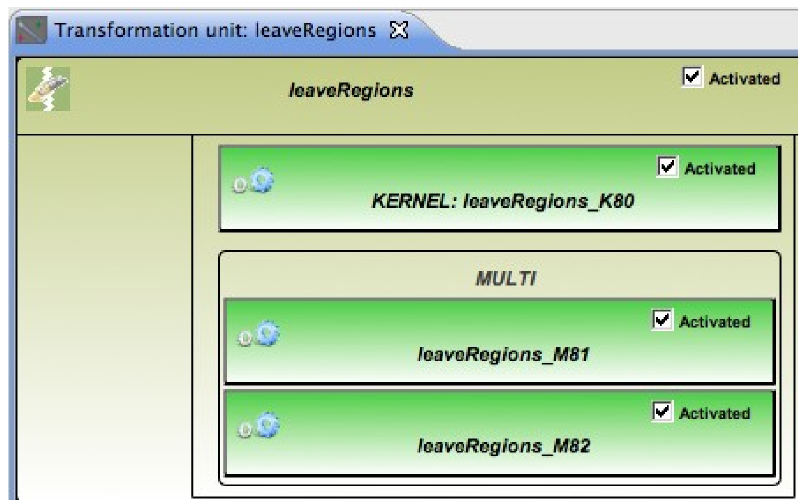
Amalgamation-Unit *leaveState2*

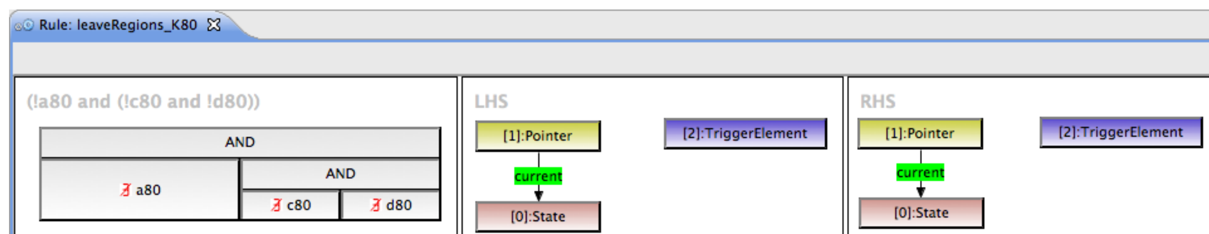


Diese Unit löscht alle *current*-Kanten von Unterzuständen mit einem nicht aktiven Oberzustand. Dies bedeutet, wenn ein Zustand deaktiviert wird, werden auch alle seine Unterzustände deaktiviert.



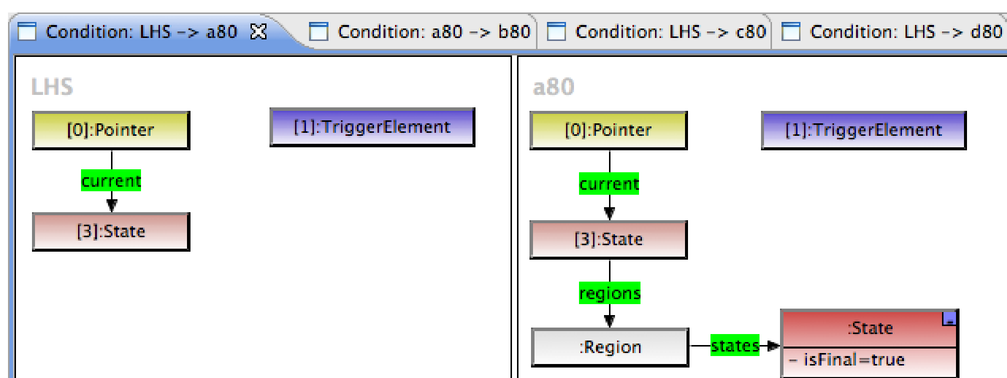
Amalgamation-Unit *leaveRegions*

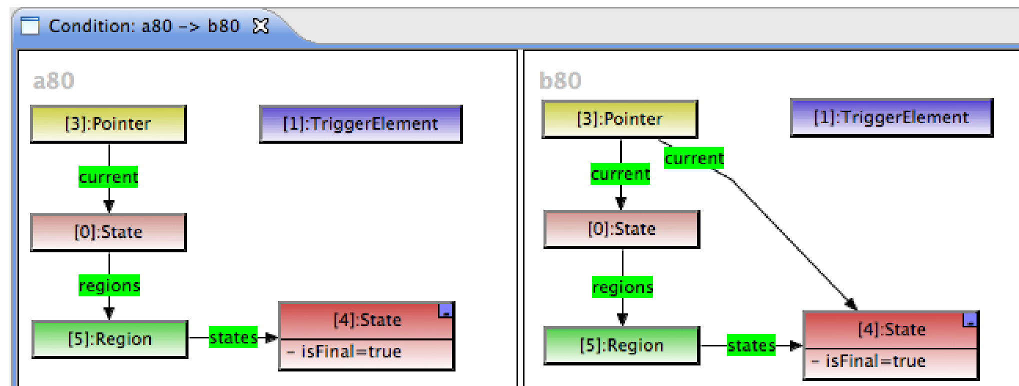




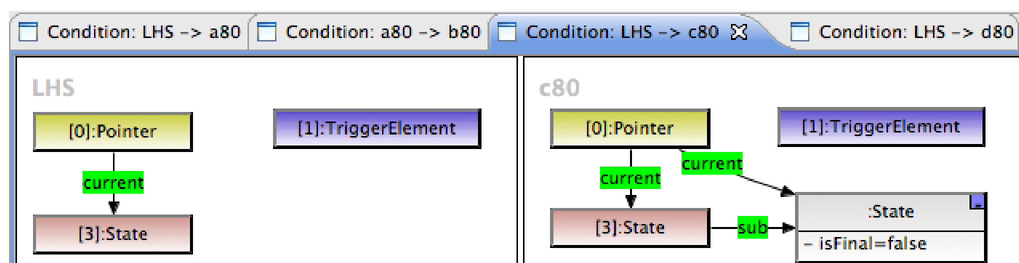
Die Kernregel *leaveRegions_K80* führt keine Transformation durch. In der Anwendungsbedingung der Kernregel werden einige Kriterien vor der Transformation durch ihre Multiregeln sichergestellt: $LHS \rightarrow (\nexists a80 \rightarrow \nexists b80) \quad \nexists c80 \quad \nexists d80$

1. Alle Unterzustände eines aktiven Zustandes mit *isFinal* = „*true*" müssen ebenfalls aktiv sein.

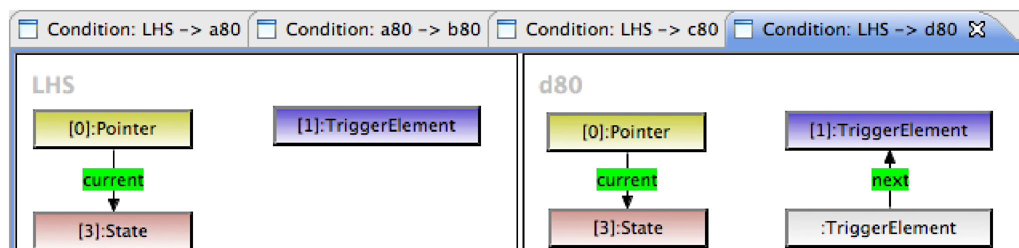




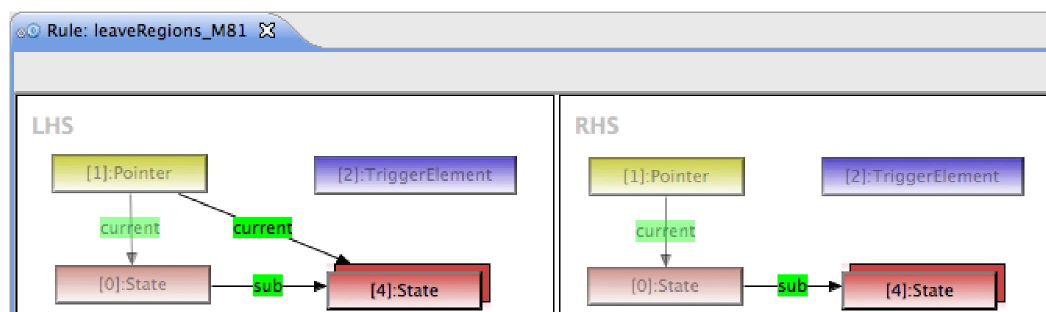
2. Die Unterzustände eines aktiven Zustandes mit *isFinal* = „*false*" dürfen nicht aktiv sein.



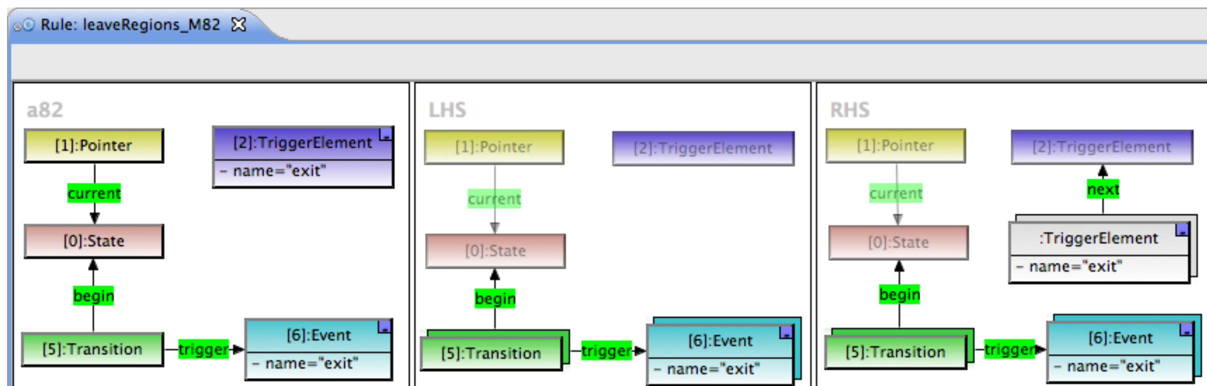
3. Das aktuelle Triggerelement darf kein Nachfolger eines anderen Triggerelements sein.



Die Multiregel *leaveRegions_M81* deaktiviert alle aktiven Unterzuständen. Dies bedeutet, dass alle *current*-Kanten zu aktiven Unterzuständen gelöscht werden.

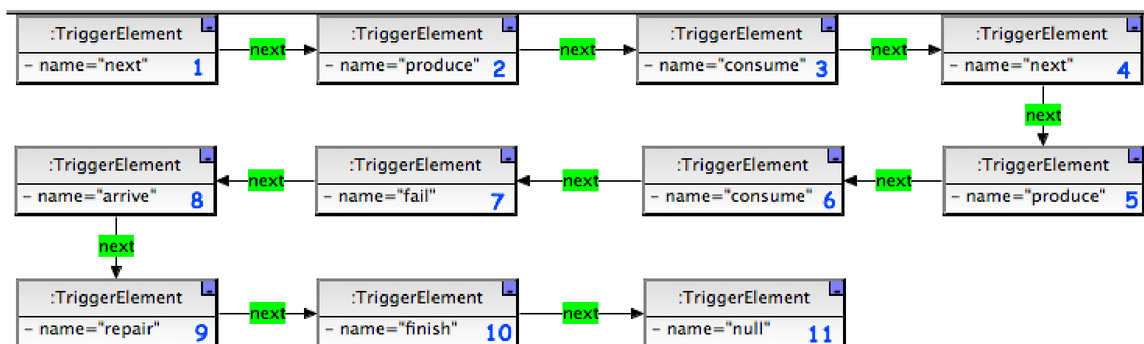


Die Multiregel *leaveRegions_M82* fügt für jedes Ereignis mit dem Namen *exit* ein gleichnamiges Triggerelement ein. Diese Regel kann nur angewendet werden, wenn das aktuelle Triggerelement nicht bereits den Namen *exit* hat.

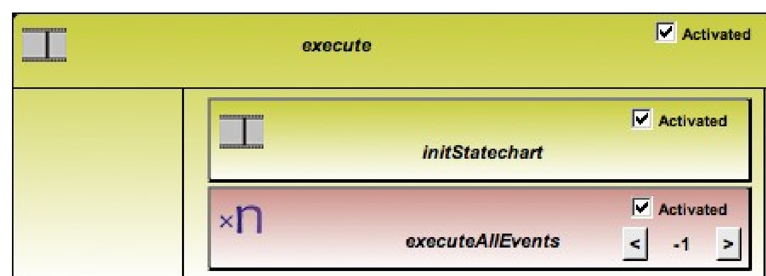


7.4 Statechart-Simulation

Wie im Abschnitt 7.3.1 erwähnt wurde, müssen Triggerelemente im Startgraphen (Abbildung 59) definiert werden, um den Ablauf vom ProdCons-Beispiel zu bestimmen.



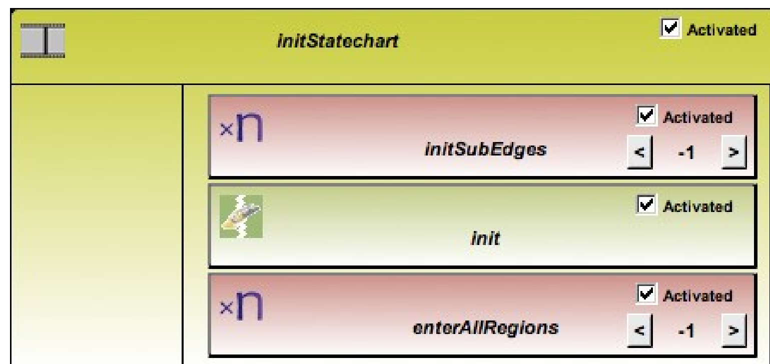
Für die komplette Automatisierung der Simulationsausführung dient die Sequential-Unit *execute*. Sie enthält zwei Units, die für die Initialisierung (*initStatechart*) und für das Ausführen von Ereignissen (*executeAllEvents*) sorgen.



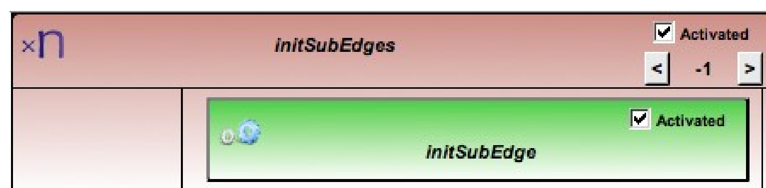
Eine detaillierte Information über verschiedene Arten von Transformation-Units sind in *[Sch2010]* und *[BESW10]* zu finden.

7.4.1 Die Sequential-Unit *initStatechart*

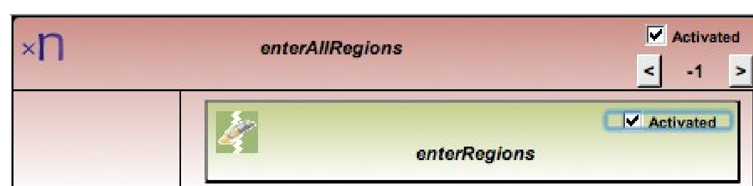
Diese Unit besteht aus drei sequentiell auszuführenden Tranformation-Units, die für die Initialisierung zuständig sind.



1. Die Counted-Unit *initSubEdges* führt die Regel *initSubEdge* solange aus, bis alle *sub*-Kanten eingefügt sind.



2. Die Amalgamation-Unit *init* wird einmal ausgeführt. Im ProdCons-Beispiel bedeutet das, dass die Verbindung zwischen der *StateMachine* und dem *Pointer* hergestellt wird und der *Pointer* auf den *prod*-Zustand mit der *new*-Kante zeigt.
3. Die Counted-Unit *enterAllRegions* führt die Amalgamation-Unit *enterRegions* solange aus, bis alle Initialzustände aktiviert sind und alle *new*-Kanten durch *current*-Kanten ersetzt sind. Im ProdCons-Beispiel bedeutet das, dass nach der Ausführung der *Pointer* mit der *current*-Kante auf die Zustände *prod*, *produced*, *empty* und *wait* zeigt.

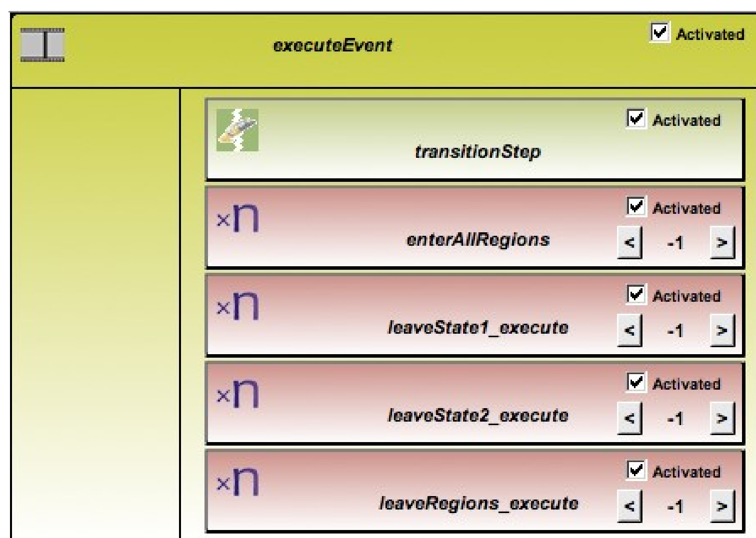


7.4.2 Die Counted-Unit *executeAllEvents*

Diese Unit steuert den Kern der gesamten Simulation. *executeAllEvents* führt die Sequential-Unit *executeEvent* so oft aus, bis alle weiter oben definierten Triggerelemente abgearbeitet sind. Zum Schluss existiert nur noch ein Triggerelement mit *name = „null“* im Arbeitsgraphen.



Die Sequential-Unit *executeEvent* enthält fünf sequentiell auszuführende Transformation-Units.



Im Folgenden wird das erste Ereignis *next* ausführlich beschrieben.

1. Die Amalgamation-Unit *transtitionStep* wird einmal ausgeführt. Die Kernregel *transitionStep_K50* findet ein Match im Arbeitsgraphen mit dem Ereignis *next*. Da alle Anwendungsbedingungen erfüllt sind (die Transition hat keinen Wächter, keine Aktion und der *produced*-Zustand hat keine Unterzustände), werden das erste Triggerlement und die *current*-Kante zwischen dem *Pointer* und dem *produced*-Zustand gelöscht und eine *new*-Kante vom *Pointer* zum *prepare*-Zustand eingefügt.

Das Ereignis *next* an dem *consumed*-Zustand gehört nicht zum Match, da dieser Zustand nicht aktiv ist.

2. Die Counted-Unit *enterAllRegions* ersetzt alle *new*-Kanten durch *current*-Kanten.
3. Die restlichen Units werden nicht ausgeführt, da kein Match gefunden wird und die Bedingungen nicht zutreffen.

Die weiteren Ereignisse werden analog ausgeführt. Daher werden sie nicht detailiert beschrieben. Stattdessen werden die aktuellen Zustände vor und nach der Ausführung der entsprechenden Ereignisse tabellarisch aufgelistet. Am Anfang hat der Startgraph folgende aktuelle Zustände: *prod*, *produced*, *empty*, *wait*.

Durchzuführende Ereignise	aktuelle Zustände nach der Ausführung	auszulösende Aktion	aktuelle Zustände nach der Ausführung
next	*prod, prepare, empty, wait*		
produce	*prod, produced, empty, wait*	*incbuff*	*prod, produced, full, wait*
consume	*prod, produced, full, consumed*	*decbuff*	*prod, produced, empty, consumed*
next	*prod, prepare, empty, wait*		
produce	*prod, produced, empty, wait*	*incbuff*	*prod, produced, full, wait*
consume	*prod, produced, full, consumed*	*decbuff*	*prod, produced, empty, consumed*
fail	*error, call*		
arrive	*error, repair*		
repair	*error, repair*		
finish	*error*	*exit*	*prod, produced, empty, wait*

8 Verwandte Arbeiten

Dieses Kapitel präsentiert die Arbeiten, die mit dem Thema der vorliegenden Diplomarbeit verwandt sind. Im Umfeld von Graphtransformationen gibt es zahlreiche Arbeiten. Diese lassen sich in grob in Werkzeuge zur Graphtransformation und solche für Modelltransformationen unterteilen. In diesem Kapitel werden vier ausgewählte Arbeiten kurz vorgestellt. Am Ende des Kapitels folgt eine Liste mit weiteren Arbeiten.

8.1 ATL

ATL (Atlas Transformation Language) ist eine deklarative und imperative Transformationssprache. Sie dient der Umformung von Modellen. ATL wurde als Eclipse-Plug-in entwickelt und bietet einen Editor mit Standardwerkzeugen an, wie Syntax-Highlighting und Debugger, die die Entwicklung von ATL-Transformationen erleichtern. Einen visuellen Editor für ATL gibt es zur Zeit noch nicht.

Der Transformationsprozess in ATL wird als Modul beschrieben. Ein Modul beinhaltet einen Header, einen Import, einen Helper und mehrere Regeln.

1. Der Header deklariert das Quell- und Zielmodell.
2. Mit Importanweisungen können Bibliotheken importiert werden.
3. Helper definieren globale Variablen und Hilfsfunktionen.
4. Eine ATL-Transformation basiert auf mehreren Regeln. Die Regeln stellen dar, wie Elemente im Quellmodell ausgewählt werden und wie die Initialisierung und Erstellung von Elementen des Zielmodells erfolgt.

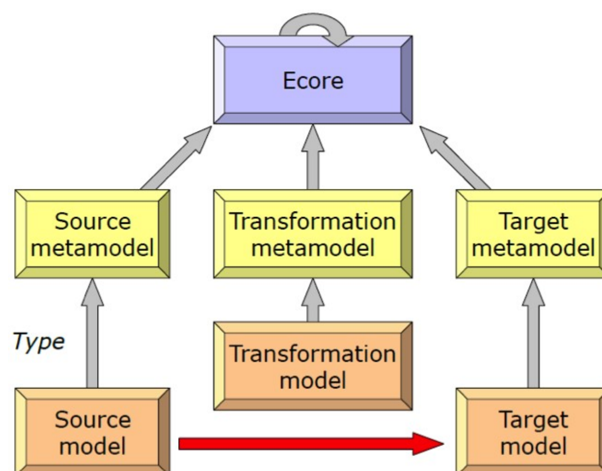


Abbildung 60: Source-Target-Ausführung [BEK0910]

Bei ATL handelt es sich um eine Source-Target-Ausführung. Dies bedeutet, dass das Zielmodell aus einem Read-Only-Quellmodell generiert wird. Das Quellmodell bleibt bei der Transformation unberührt.

8.2 AGG

AGG (Attributed Graph Grammar) ist eine regelbasierte visuelle Sprache für Graphtransformationen mit folgenden Merkmalen: *[AGG]*

- Komplexe Datenstrukturen werden als typisierte Graphen modelliert.
- Das Systemverhalten wird durch graphische Regeln mithilfe von IF-THEN-Bedigungen festgelegt.
- Darüber hinaus können AGG-Regeln über negative Anwendungsbedingungen verfügen, mit dem die Nicht-Existenz von Teilgraphstrukturen ausgedrückt werden kann.
- Graphen werden durch die Anwendung von Regeln transformiert.
- Die sequentielle Anwendung von Regeln zeigt ein Anwendungsszenario an.
- AGG-Graphen können durch Java-Objekte und -Typen attributiert werden.
- Regeln können durch Java-Ausdrücke attributiert werden, die während der Regelanwendung ausgewertet werden.

Wie auch der Henshin-Editor bietet die AGG-Entwicklungsumgebung eine graphischen Benutzeroberfläche mit mehreren visuellen Editoren, einem Interpreter und einer Reihe von Validierungstools.

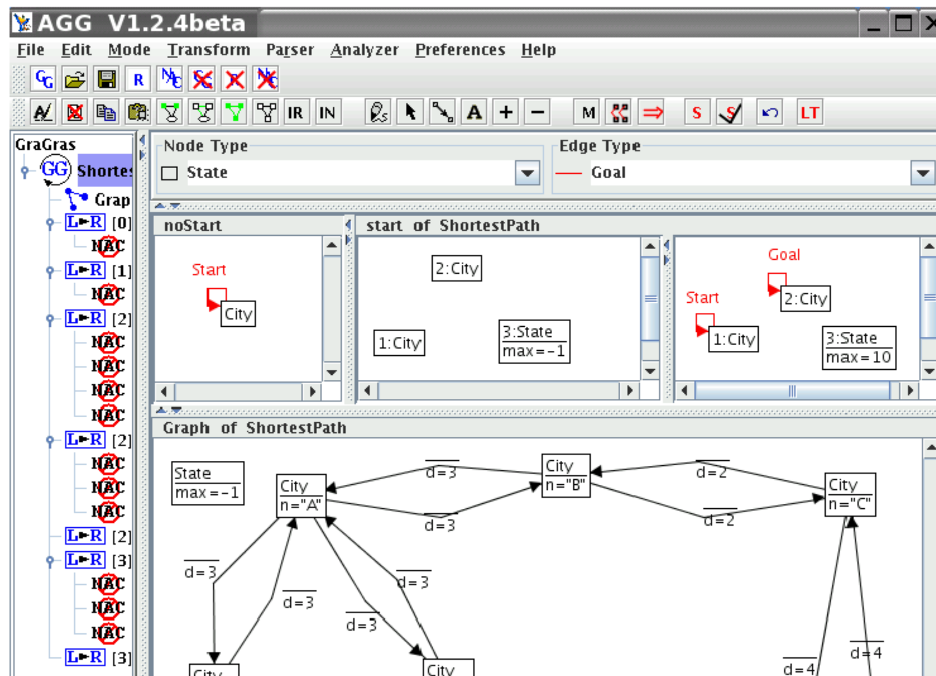


Abbildung 61: AGG - Benutzeroberfläche

8.3 Fujaba

Fujaba steht für "From UML to Java and back again". Es ist ein UML-Werkzeug, das die Unterstützung von der modellbasierten Softwareentwicklung und dem Reengineering bietet. Es wurde in der Programmiersprache Java geschrieben und ist daher plattformunabhängig. *[WP-FUJ10, Fuj10]*

Das Fujaba-Projekt wurde im Jahr 1997 an der Universität Paderborn ins Leben gerufen und wird inzwischen von mehreren Hochschulen in ganz Deutschland und in einigen anderen Ländern entwickelt. Im Jahr 2002 wurde Fujaba neu konzipiert als ein Werkzeug mit einer Plug-In Architektur, die das Hinzufügen von Funktionalität ermöglicht. Seit 2006 gibt es eine Fujaba Integration in die Eclipse-Plattform (Fujaba4Eclipse). Die Entwicklung der visuellen Editoren in Fujaba4Eclipse basiert auf GEF und EMF.

Die Kernfunktionen von Fujaba sind:

- Modellierung von Objektstrukturen mit UML-Klassendiagrammen
- Modellierung von Verhalten einzelner Methoden mit Story-Diagrammen (eine Kombination von UML-Aktivitätsdiagrammen und der Graphtransformationssprache „Story Pattern“)
- Generieren von Quellcode auf Basis formaler, grafisch definierter Modelle

8.4 VIATRA2

VIATRA2 ist eine Abkürzung für VIsual Automated model TRAnsformations. VIATRA2 stellt eine regel- und musterbasierte Transformationssprache zur Manipulation von EMF-Modellen unter Verwendung von Graphtransformationsprinzipien kombiniert mit abstrakten Zustandsmaschinen zur Verfügung. Diese Sprache bietet erweiterte Konstrukte zur Abfrage und zum Manipulieren von Modellen (z.B. generische Transformation und Meta-Transformationsregeln). *[Via10]*

8.5 WeitereArbeiten

Weitere Werkzeuge zur Graphtansformation, die hier nicht weiter beschrieben werden, sind:

- MOMENT *[Bor07]*
- PROGRES *[Pro99]*
- MoTMoT *[Mot07]*
- EWL *[EWL07]*
- GReAT *[Gre07]*
- GrGen.NET *[GK08]*

Bis auf AGG unterstützt keiner der bisher existierenden Modelltransformationsansätze eine Konfluenz und eine Terminierungsanalyse der EMF-Transformationsregeln. Hier dienen der Henshin-Ansatz und die Werkzeugumgebung als eine Brücke zu etablierten Werkzeugen und formalen Techniken zur Graphtransformationen für modellgetriebene Entwicklung auf Basis von EMF.

Nach Abschluss dieser Arbeit ist Henshin das einzige graphische Entwicklungswerkzeug für EMF-Modelltransformationen, das auf Graphtransformationen beruht und komplexe Anwendungsbedingungen, Transformation Units und die Anwendung auf Multiobjektstrukturen mit amalgamierten Regeln unterstützt.

9 Zusammenfassung und Ausblick

In diesem Abschnitt werden die Ergebnisse der Weiterentwicklung des Henshin-Editors zusammengefasst und abschließend einen kurzen Ausblick über geplante Erweiterung im Henshin-Editor gegeben.

9.1 Zusammenfassung

Der Henshin-Editor mit den erweiterten Funktionalitäten erfüllt die im Kapitel 4 gestellten Anforderungen. Es ist jetzt möglich, auch komplexe Anwendungsbedingungen visuell zu definieren. Darüber hinaus unterstützt der Henshin-Editor nun auch geschachtelte Anwendungsbedingungen und beide Varianten von *Application Conditions* (*PAC* und *NAC*). All diese Erweiterungen führen zu genaueren und aussagekräftigeren Definitionen von Anwendungsbedingungen.

Graphtransformationen können durch die Realisierung von Amalgamation-Units nun an mehr als einem Match durchgeführt werden. Amalgamation-Units ermöglichen eine variable Anzahl von Transformationen, ohne dass diese in der Transformationsregel vorher festgelegt werden müssen. Diese Erweiterung bietet Nutzern mehr Flexibilität beim Definieren von Transformationsregeln.

Durch die im Rahmen dieser Arbeit entstandenen Erweiterungen und auch die von *[Sch10]* sind nun alle EMF-Modellinstanzen im Henshin-Editor verfügbar. Sie können erstellt, bearbeitet und gelöscht werden. Daher sind weitere Erweiterungen des visuellen Editors zur Zeit nicht geplant.

9.2 Ausblick

Aktuell laufende Arbeiten befassen sich mit der formalen Analyse von Verhaltensmodellen, die mit Henshin erstellt wurden. Sie basiert auf die Theorie von Graphtransformationssystemen. Die Analyse von Modelleigenschaften wie Termination und Konfluenz wird zur Zeit mit AGG durchgeführt.

Als weitere Entwicklung des Henshin-Editors ist es wünschenswert, die Analyseresultate geeignet im graphischen Henshin-Editor darzustellen und damit dem Modellierer Hinweise auf Modell-Inkonsistenzen zu geben.

A Literaturverzeichnis

[AGG] http://user.cs.tu-berlin.de/~gragra/agg/agg-docu.html

[BEEG10] E. Biermann, H. Ehrig, C. Ermel, U. Golas. *A Visual Interpreter Semantics for Statecharts Based on Amagamated Graph Transformation*, 2010.

[BEK+ 06] E. Biermann, K. Ehrig, C. Ko hler, G. Kuhns, G. Taentzer, E. Weiss.*Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework.* In Nierstrasz et al. (eds.), Proc. of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'06). LNCS 4199, pp. 425–439. Springer, Berlin, 2006.

http://tfs.cs.tu-berlin.de/publikationen/Papers06/BEK+06a.pdf

[BEK0910] E. Biermann, C. Ermel, S. Krause. *Modelltransformation mit Henshin und ATL*, 2009/2010

[BEL+ 10] E. Biermann, C. Ermel, L. Lambers, U. Prange, G. Taentzer. *Introduction to AGG and EMF Tiger by Modeling a Conference Scheduling System.* Software Tools for Technology Transfer, 2010. To appear. http://www.springerlink.com/content/p4n1g45627852743/

[BESW10] E. Biermann, C. Ermel, J. Schmidt, A. Warning. *Visual Modeling of Controlled EMF Model Transformation using Henshin.* Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010), 2010.

[BET08] E. Biermann, C. Ermel, G. Taentzer. *Precise Semantik of EMF Modell Transformations by Graph Transformation.* Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS´08). : Springer, 2008. 5301, 53-67.

[Bor07] A. Boronat. MOMENT: *A Formal Framework for Model Management*. PhD theses. : Universität Politecnica de Valencia, 2007

[ECL10] http://www.eclipse.org/modeling/emft/henshin/

[EBM08] E. Claudia, B. Enrico, M. Tony. *Das Graphische Editing Framework (GEF) von Eclipse Teil 1: Architektur*. Vorlesungsfolie des Visuelle-Sprachen-Projekt, 2008

[EBM09] E. Claudia, B. Enrico, M. Tony. *Einführung in das Eclipse Modeling Framework (EMF)*. Vorlesungsfolie des Visuelle-Sprachen-Projekt, 2009

[EHL10b] H. Ehrig, A. Habel, L. Lambers. *Parallelism and Concurrency Theorems for Rules with Nested Application Conditions.* Electr. Communications of the EASST 26, 2010. http://journal.ub.tu-berlin.de/index.php/eceasst/issue/view/36

[Eb2010] Stand 8. Dezember 2010. http://www.ralfebert.de/rcpbuch/overview/

[EMF10] Stand 15. November 2010. http://www.eclipse.org/modeling/emf/

[EMT09] TFS-Group, TU Berlin. EMF Tiger. 2009. http://tfs.cs.tu-berlin.de/emftrans.

[Fuj10] Stand 6. November 2010. http://www.fujaba.de/no_cache/home.html

[GK08] R. Geiß, M. Kroll. GrGenNET: *A Fast, Expressive, and General Purpose Graph Rewrite Tool.* Proc 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'07).: Springer, 2008.

[Gre10] Stand 12. November 2010. http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT

[Har1987] D. Harel, *Statechart: A Visual Formalism fpr Complex Systems Science of Computer Programming*, 1987

[Kru2000] P. Kruchten, *The Rational Unified Process an Introduction Second Edition*, Addison-Wesley-Longman, 2000.

[Mot10] FOTS-Group. University of Antwerp. MoTMoT: Model driven, Template based, Model Transformer. [Online] 2010. http://www.fots.ua.ac.be/motmot/index.php

[Pfl01] S.L. Pfleeger, *Software Engineering: Theory and Practice (2nd Edition)*, Prentice Hall, 2001.

[Pro99] A. Schürr, A. Winter, A. Zündorf. The PROGRES-Approach: *Language and Environment. Handbook of Graph Grammars and Computing by Graph Transformation*, Volume 2: Applications, Languages and Tools. : World Scientific, 1999, S. 487 – 550.

[RD01] C. Rupp, J. Dallner, *Mustergültige Anforderungen*, OBJEKTspektrum, Nr. 3, S. 32 – 37, 2001.

[Sch10] J. Schmidt, *Entwicklung eines visuellen Editors zur Steuerung von EMF-Modelltransformation*, Bachelor Arbeit, 2010.

[Via10] Stand 12. November 2010. http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html#VIATRA2Area

[WP-AMAL10] Stand 1. November 2010. http://de.wikipedia.org/wiki/Amalgam

[WP-EMF10] Stand 15. November 2010. http://de.wikipedia.org/wiki/Eclipse_Modeling_Framework

[WP-FUJ10] Stand 6. November 2010. http://de.wikipedia.org/wiki/Fujaba

[WP-ZUS10] Stand 12. Dezember 2010. http://de.wikipedia.org/wiki/Zustandsdiagramm

B Benutzerhandbuch

Dieser Anhang beinhaltet das vollständige Benutzerhandbuch des Henshin-Editors.



Henshin-Editor:

Visuelle Entwicklungsumgebung für komplexe EMF-Modelltransformationen

Benutzerhandbuch

Johann Schmidt

Angeline Warning

Für den Henshin-Editor wird folgende Software vorausgesetzt:

- **Java** Version 1.6
- **Eclipse** Version 3.6
- **EMF - Eclipse Modeling Framework SDK** Version 2.6.0
- **Graphical Editing Framework GEF SDK** Version 3.6.0
- **MuvitorKit**
- **Henshin SDK**

Inhaltsverzeichnis

1 Anlegen eines Projektordners

Bevor ein Transformationssystem erstellt werden kann, wird ein neuer Projektordner in der Eclipse-Entwicklungsumgebung benötigt.

Klicken Sie auf *File* → *New* → *Project...*.

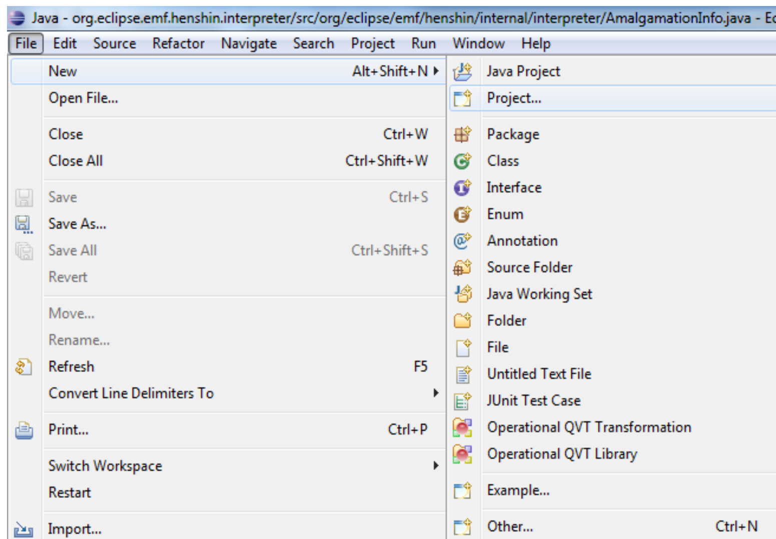


Abbildung 1

Öffnen Sie als nächstes den Ordner *General* und wählen Sie *Project* aus. Klicken Sie anschließend auf *Next*.

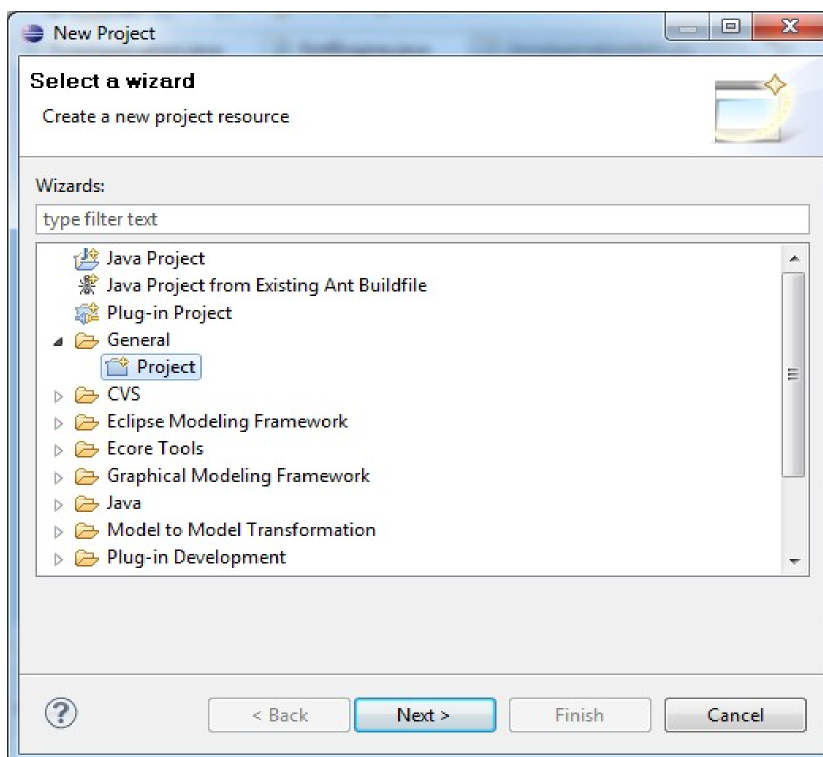


Abbildung 2

Hier können Sie einen Namen für das Projekt eingeben. Klicken Sie danach auf *Finish.*

Abbildung 3

2 Erstellen eines Transformationssystems

Nach Anlegen eines Projektordners kann ein Transformationssystem erstellt werden.

Klicken Sie hierzu mit der rechten Maustaste auf den zuvor erstellten Projektordner. Wählen Sie *New → Other....*

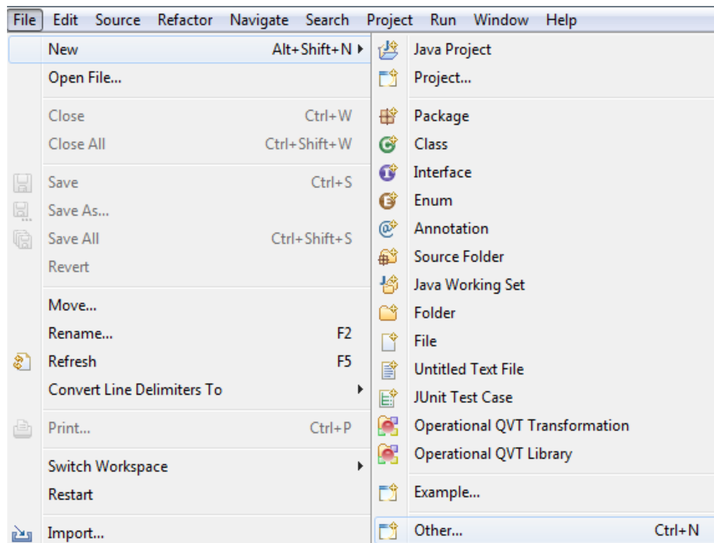


Abbildung 4

Öffnen Sie den Ordner *Other* und wählen Sie *Henshin File Creation Wizard* aus. Klicken Sie danach auf *Next*.

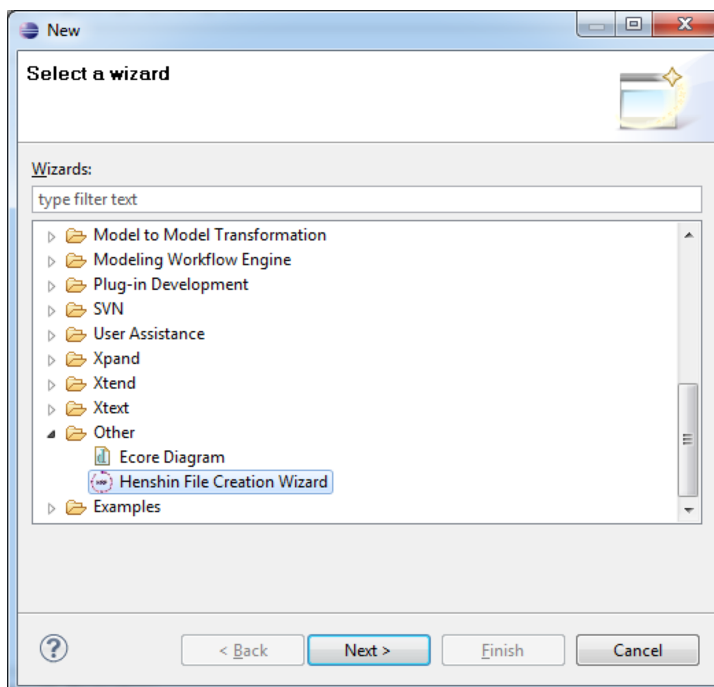


Abbildung 5

Hier können Sie einen Namen für das Transformationssystem vergeben oder den vorgegebenen Namen beibehalten. Durch einen Klick auf *Finish* wird das Transformationssystem erstellt.

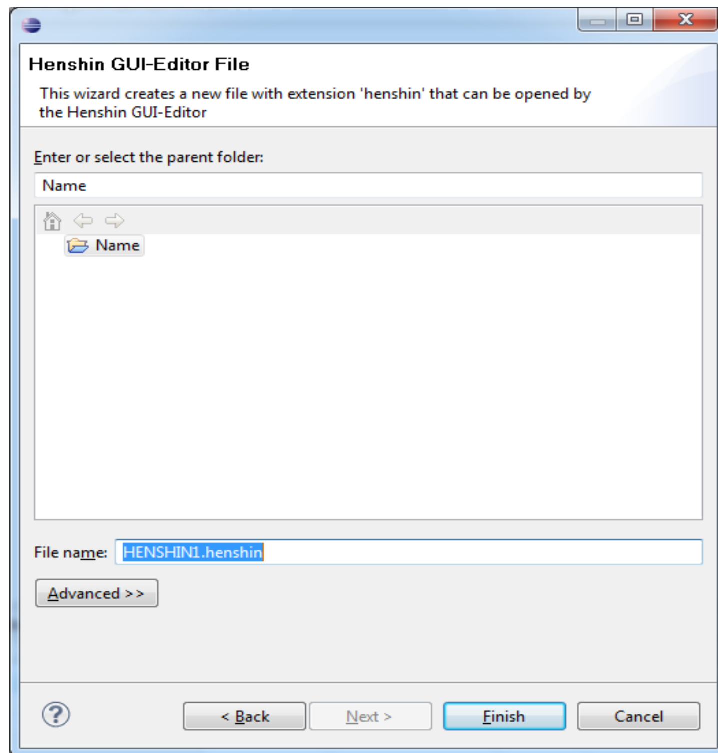


Abbildung 6

Wurde alles korrekt ausgeführt, sehen Sie automatisch die *Henshin Editor*-Ansicht.

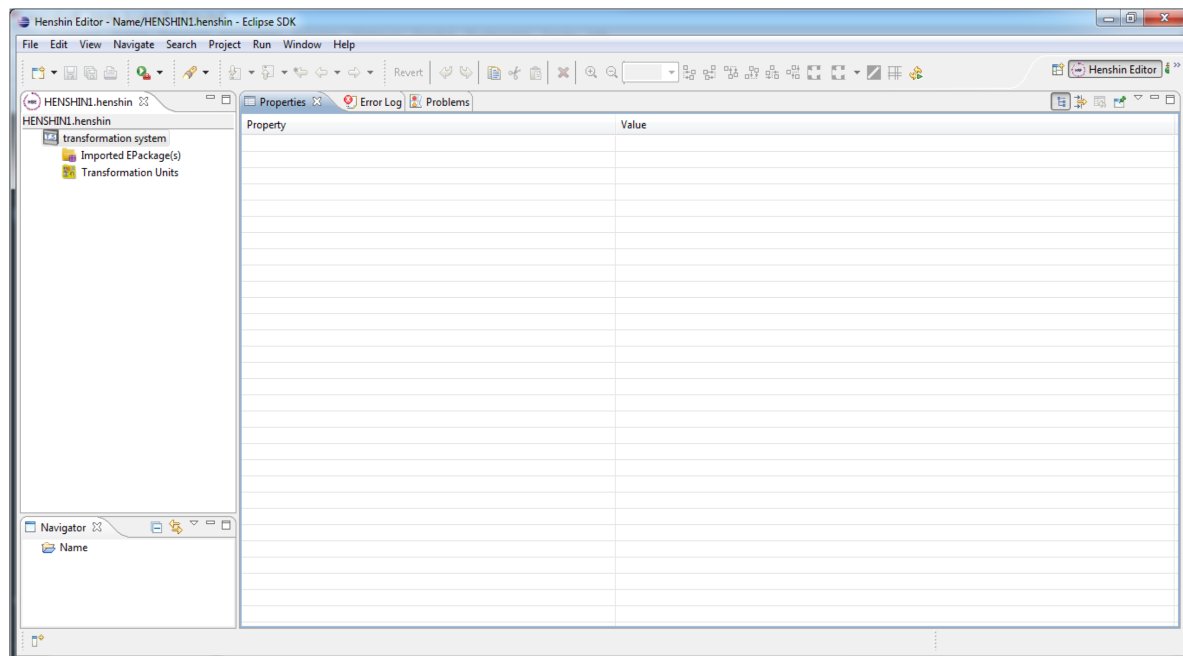


Abbildung 7 : *Henshin Editor*-Ansicht

3 Importieren eines EMF-Modells

Nachdem die neu erstellte Henshindatei geöffnet ist, können ein oder mehrere EMF-Modelle importiert werden.

Wählen Sie mit einem rechten Mausklick auf *transformation system* den Punkt *Import EMF Package* aus.

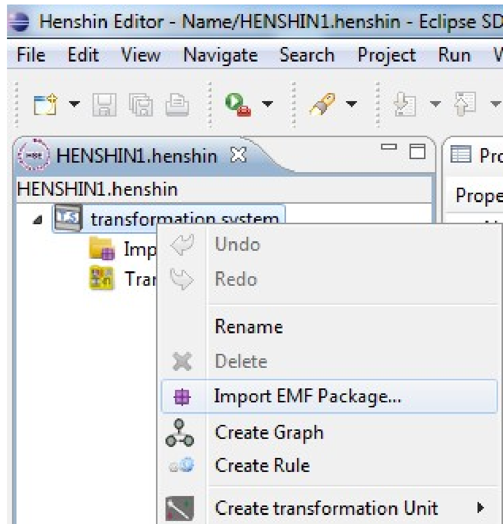


Abbildung 8: Transformationssystem-Kontextmenü

Die vorhandenen EMF-Modelle im Workspace werden Ihnen in einer Liste angezeigt. Wählen Sie ein Model aus der Liste aus oder importieren Sie eine Ecore-Datei, in dem Sie erst die Schaltfläche *Workspace…* betätigen und dann *File System* wählen.

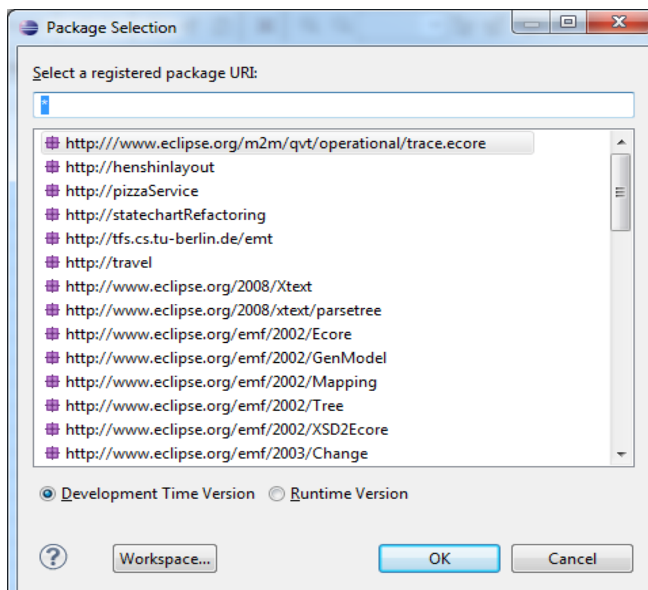


Abbildung 9: Auswahl der EPackages

Beispiel: Wir wählen das *pizzaService* Modell aus und klicken auf *OK*. Importierte Modelle werden in der Baum-Ansicht unter *ImportedEPackage(s)* angezeigt.

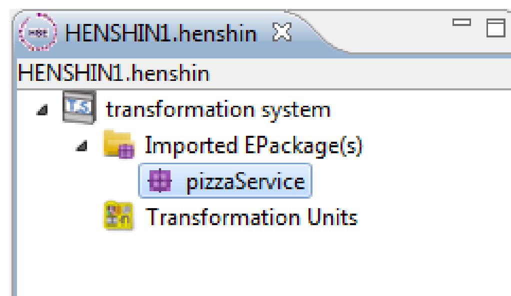


Abbildung 10

4 Graphen erzeugen

Um einen neuen Graph zu erzeugen, wählen Sie im Kontextmenü von *transformation system* den Punkt *Create Graph* aus, (Abbildung 8).

Geben Sie den Namen des Graphen ein oder behalten Sie den vorgeschlagenen Namen und klicken Sie auf *OK.*

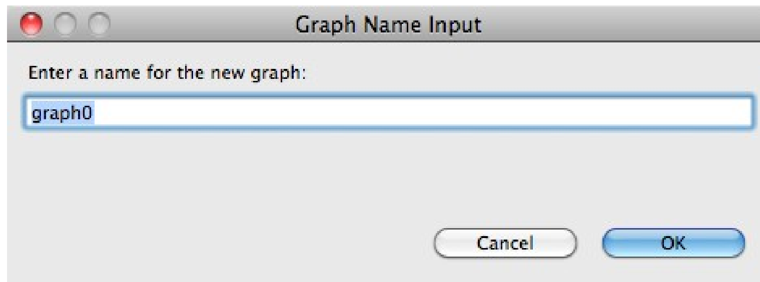


Abbildung 11

Die Namen aller Graphen eines Transformationsystems müssen eindeutig sein. Der Henshin-Editor stellt dies sicher, indem bereits vorhandene Namen zurückgewiesen werden. Nach Bestätigung des Dialogs, wird die graphische Ansicht des neuen Graphen geöffnet.

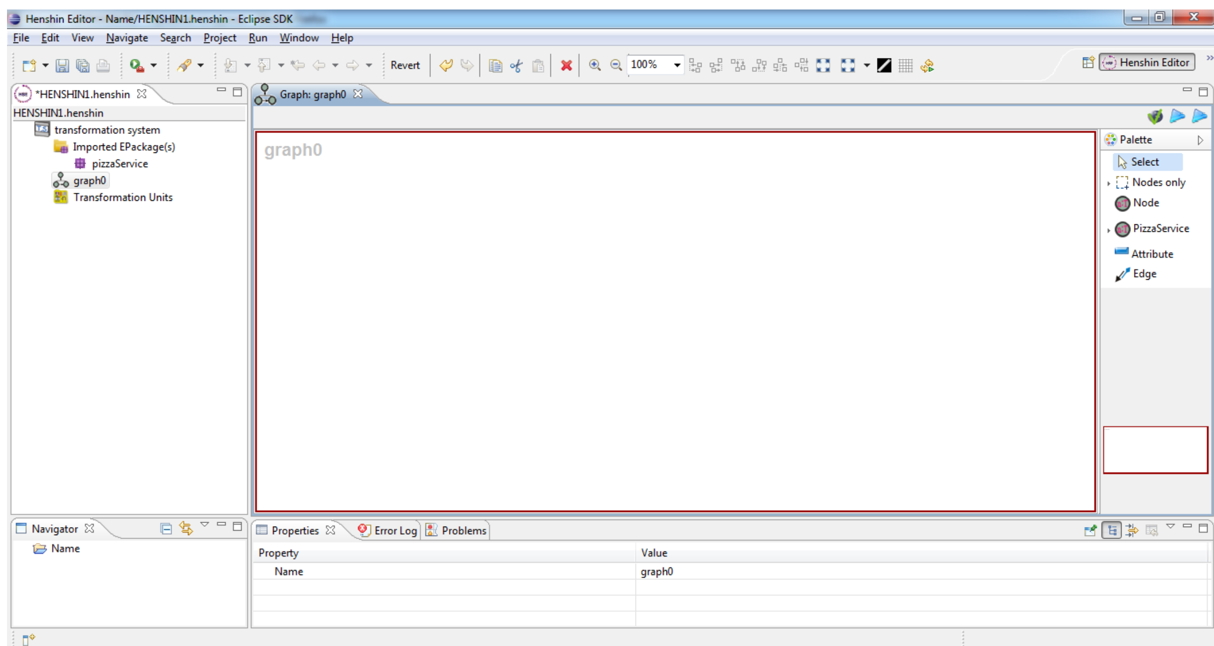


Abbildung 12: Graph-Ansicht (Graph View)

4.1 Knoten erzeugen

Neue Knoten können über das Kontextmenü eines Graphen in der Baumansicht oder über die Palette erstellt werden.

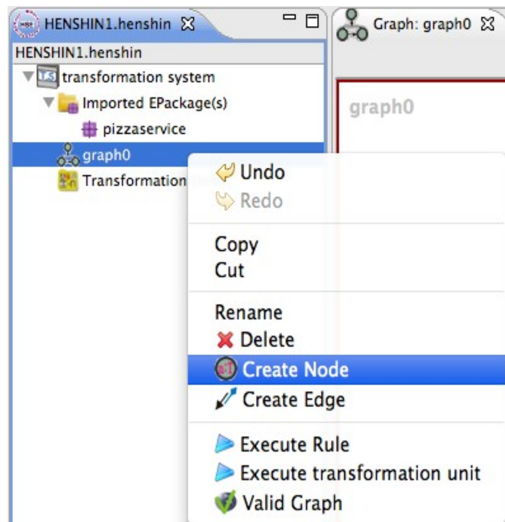


Abbildung 14: Graph-Kontextmenü

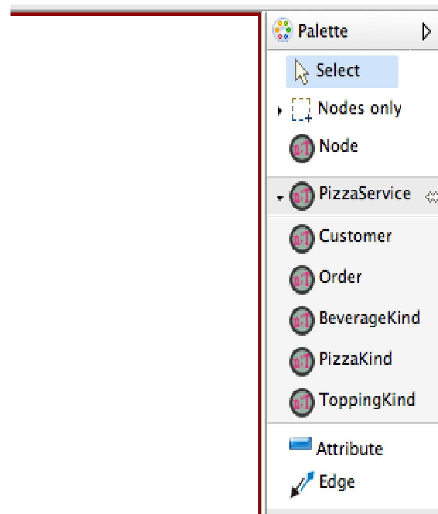


Abbildung 13: Graph-Palette

Um einen Knoten mit Hilfe des Kontextmenüs zu erstellen, wählen Sie dort den Menüpunkt *Create Node* aus. Es erscheint ein Dialog zum Bestimmen des Knotentyps.

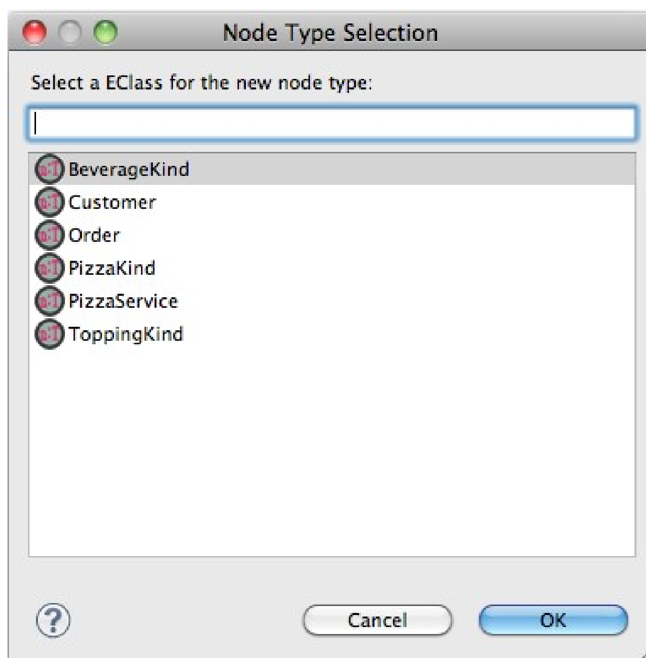


Abbildung 15: Auswahl der Knotentypen

Mit Hilfe der Palette können Sie den Knotentyp sofort festlegen. Klicken Sie hierzu zuerst den Typ in der Palette an (Abbildung 13) und danach auf die gewünschte Position in der Graphansicht. Alternativ können Sie in der Palette auch *Node* wählen. In diesem Fall erscheint nach der Festlegung der Position in der Graphansicht der Dialog zur Bestimmung des Knotentyps (Abbildung 15).

Beispiel: Wir erstellen zwei Knoten, einen vom Typ *Customer* und den anderen vom Typ *Order*.

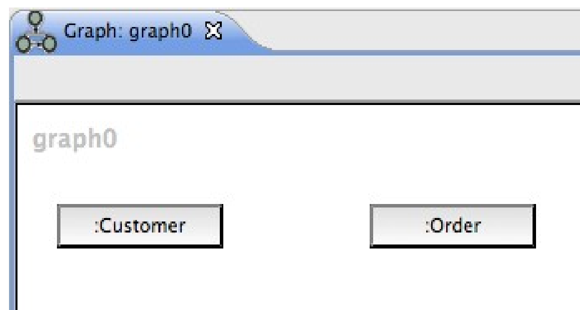


Abbildung 16

4.2 Kanten erzeugen

Das Erzeugen von Kanten erfolgt analog der Erzeugung von Knoten: Entweder über den Menüpunkt *Create Edge* des Kontextmenüs eines Graphen (Abbildung 14: Graph-Kontextmenü) oder über die Schaltfläche *Edge* in der Palette (Abbildung 13: Graph-Palette).

Wenn eine Kante über das Kontextmenü erstellt wird, werden die notwendigen Angaben in einem Dialogfenster abgefragt. Nach der Auswahl eines *Source*-Knotens erscheinen in der Spalte Target die dazu passenden *Target*-Knoten. Nach der Auswahl des *Target*-Knotens werden in der Spalte *Type* die möglichen Kantentypen angezeigt. Erst nachdem der *Source*-Knoten, der *Target*-Knoten und der Kantentyp festgelegt sind, kann der Dialog über *OK* geschlossen werden.

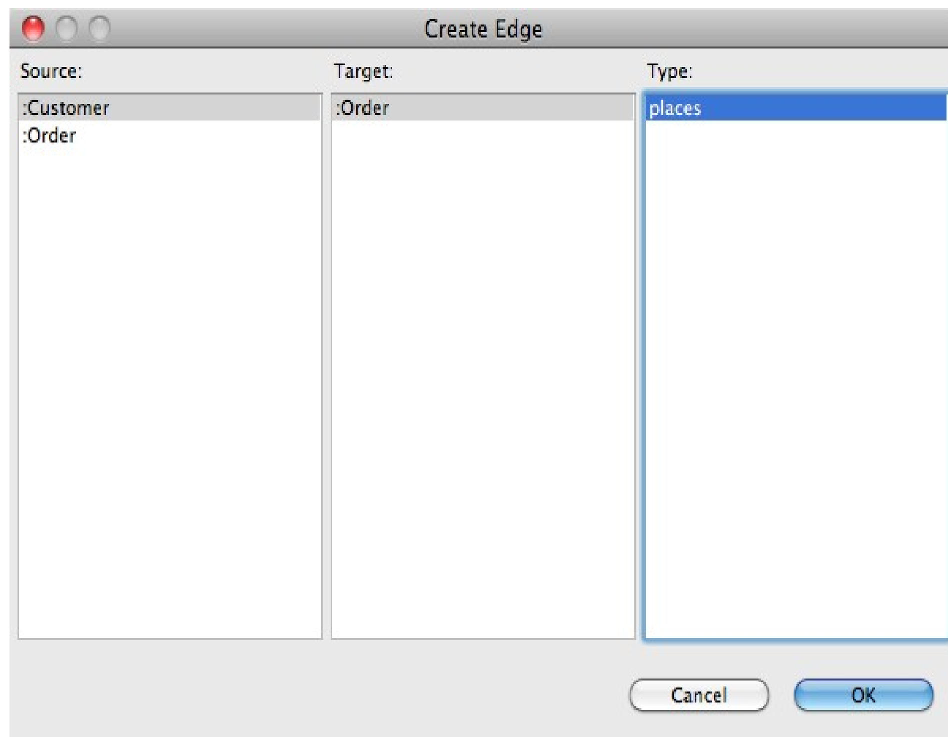


Abbildung 17: Dialog zum Erzeugen einer Kante

Die Palette bietet mehr Übersichtlichkeit, da Sie nur auf den Source- und dann auf den Target-Knoten zu klicken brauchen. Dabei werden die möglichen Kantentypen und ihre Multiplizitäten geprüft. Wenn zwischen den ausgewählten Source- und Target-Knoten mehr als einen Kantentyp existiert, wird eine Auswahl angeboten (Abbildung 19: Kantentyp Auswahl). Andernfalls wird eine Kante mit dem passenden Typ erzeugt.

Beispiel (ohne Auswahl): Erstellen Sie eine Kante mit Hilfe der Palette. Zuerst klicken Sie auf den Menüpunkt *Edge* der Palette. Danach klicken Sie erst auf den Knoten *Customer* und dann auf den Knoten *Order*. Weil nur ein Kantentyp zwischen den Knoten *Customer* und *Order* existiert, wird die Kante automatisch erstellt.

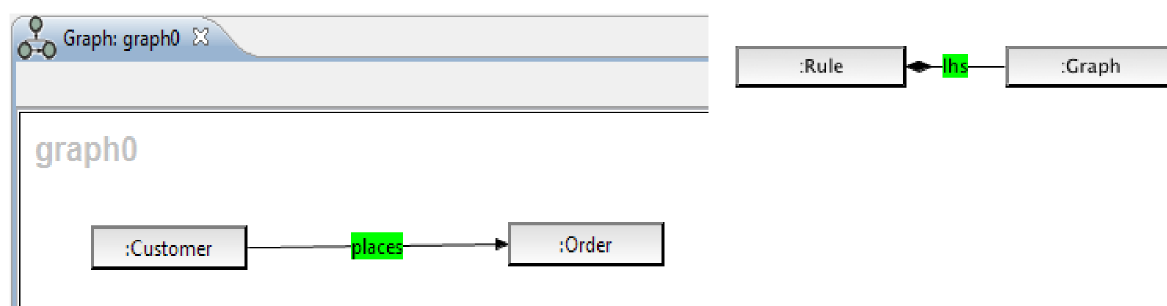


Abbildung 18

Beispiel (mit Auswahl): Importieren Sie zuerst ein Henshin-EMF-Modell. Danach erstellen Sie einen Knoten vom Typ *Rule* und einen vom Typ *Graph*. Anschließend erzeugen Sie eine *lhs*-Kante von *Rule* zu *Graph*.

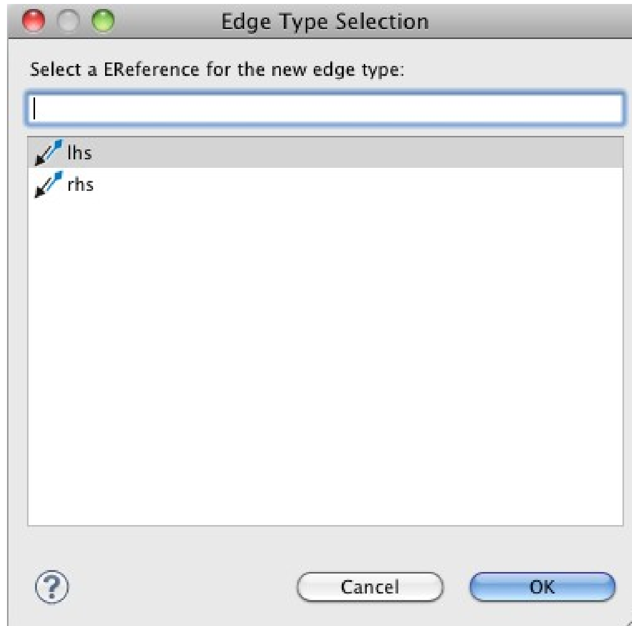


Abbildung 19: Kantentyp Auswahl

4.3 Attribute erzeugen

Knoten können um Attribute erweitert werden. Dies kann über den Menüpunkt *Create a new Attribute* des Kontextmenüs eines Knotens in der Baum- (Abbildung 20) oder in der Graphansicht (Abbildung 21) oder über die Palette (Abbildung 13, S. 112) geschehen.

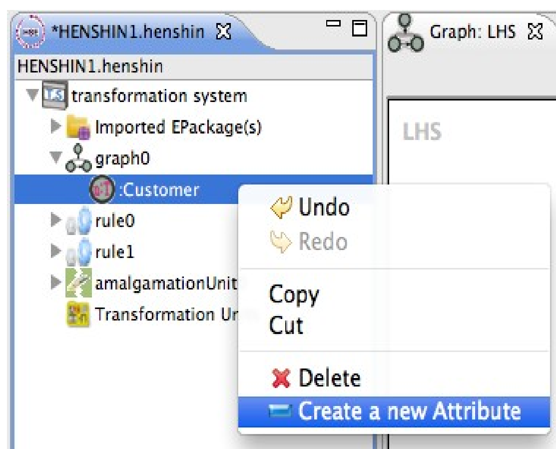


Abbildung 20: Node-Kontexmenü 1

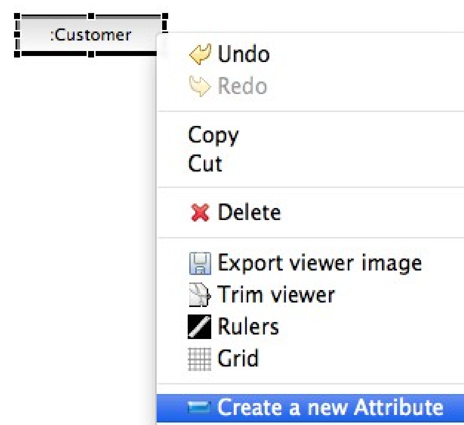


Abbildung 21: Node-Kontexmenü 2

Durch einen Klick auf die Schaltfläche *Attribute* in der Palette und danach auf den Knoten wird eine Auswahl aller möglichen Attribute mit Standard-werten in einem Dialog ange-

Abbildung 22: Dialog zum Erzeugen des Attributes

zeigt. Setzen Sie den Haken bei den Attributen, die erstellt werden sollen. Wenn nötig, ändern Sie die Werte und klicken Sie auf *Ok*.

Besitzt ein Knoten Attribute, wird dies durch das „*+*“-Symbol oben rechts angezeigt. Klicken Sie auf die Symbole „*+*“ und „-“, um die Attribute ein- und auszublenden.

Abbildung 23: Attributanzeige

4.4 Attributwerte ändern

Es gibt zwei Möglichkeiten, die Werte von Attributen zu ändern.

1. *Properties*-Ansicht

   Wählen Sie den Knoten, dessen Attribute geändert werden sollen in der Graphansicht oder in der Baumansicht aus. In der Ansicht *Properties* erscheinen daraufhin die Eigenschaften des Knotens. In der Spalte *Value* können die Attributwerte geändert werden.

2. Graphansicht

   Klicken Sie das Attribut, das geändert werden soll in der Graphansicht doppelt an. Der Wert des Attributs kann daraufhin direkt im Graphen geändert werden.

Beispiel: Ändern des Wertes des Attributes *open* auf *true*.

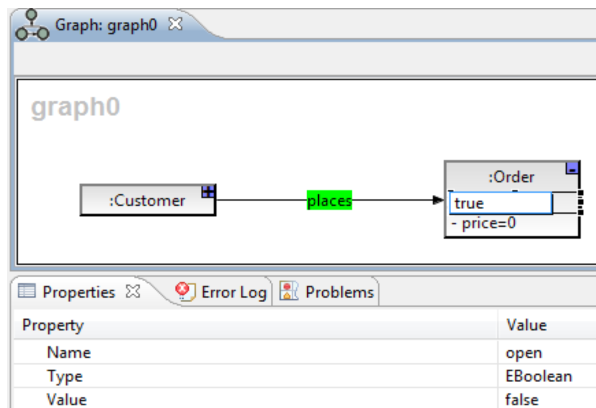


Abbildung 24: Attributwerte ändern

5 Transformationsregeln erstellen

Um eine neue Transformationsregel zu erstellen, wählen Sie im Kontextmenü von *transformation system* den Menüpunkt *Create Rule* aus. (Abbildung 8, S. 109). Geben Sie in den daraufhin erscheinenden Dialog den Namen für die Regel ein oder behalten Sie den vorgeschlagenen Namen bei und klicken auf *OK*.



Abbildung 25

Die Namen aller Regeln eines Transformationsystems müssen eindeutig sein. Der Henshin-Editor stellt dies sicher, indem bereits vorhandene Namen zurückgewiesen werden. Nach Bestätigung des Dialogs, wird die Regelansicht der neuen Regel geöffnet. Eine Regel besteht aus zwei Graphen *LHS* und *RHS* (siehe Kapitel 4.1 - 4.4). Sie können einen Graphen *LHS* in *RHS* kopieren, indem Sie in der Werkzeugleiste auf das Symbol LR (rechts oben) klicken. Dabei werden die Mappings (siehe Kapitel 5.1) zwischen den Knoten automatisch erstellt.

Beispiel: Definieren Sie eine Regel *addPizzaKind*, die für die in dem Graphen bereits existierende *Order* einen neuen Knoten *PizzaKind* erstellt.

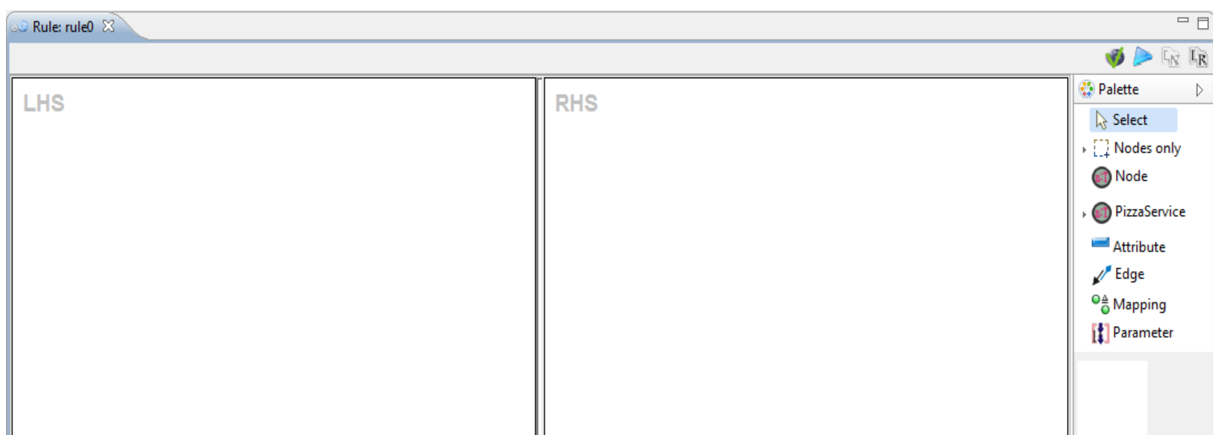


Abbildung 26: Regelansicht (Rule View)

Gehen Sie wie folgt vor:

1. Erstellen Sie in *LHS* einen Knoten vom Typ *Order*.
2. Erstellen Sie in *RHS* einen Knoten vom Typ *Order* und einen vom Typ *PizzaKind*.
3. Erzeugen Sie in *RHS* eine Kante vom Knoten *Order* zum Knoten *PizzaKind*.

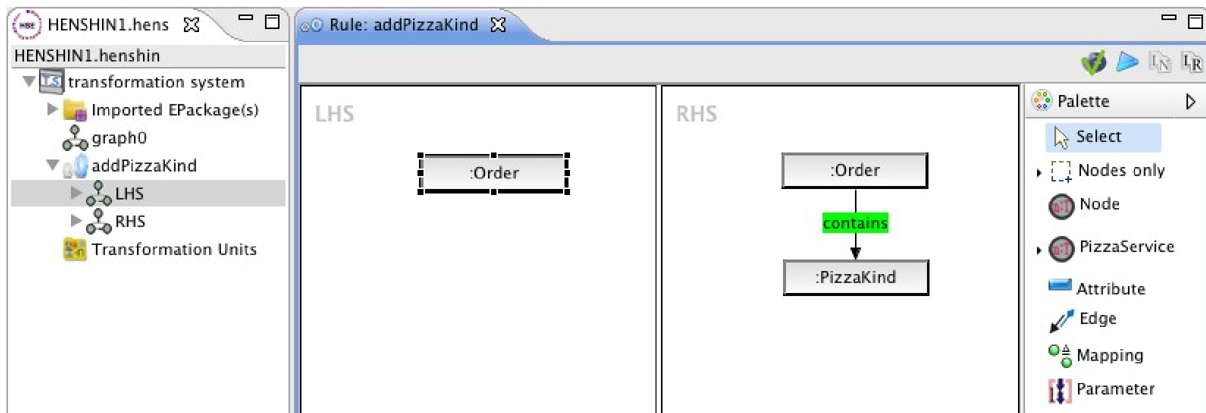


Abbildung 27: Rule *addPizzaKind*

5.1 Mapping erstellen

Soll ein Knoten in *LHS* und *RHS* denselben Knoten im Graphen darstellen (im Schnitt von *LHS* und *RHS* liegend), muss für diesen Knoten ein Mapping erstellt werden.

Sie erstellen ein Mapping, indem Sie den Punkt *Mapping* aus der Palette auswählen und dann zuerst auf den Knoten in *LHS* und danach auf den Knoten in *RHS* klicken. Gemappte Knoten erhalten dieselbe Farbe und Nummer und sind dadurch leicht als solche erkennbar.

Beispiel: Erstellen Sie ein Mapping zwischen den Knoten *Order* in *LHS* und *Order* in *RHS*.

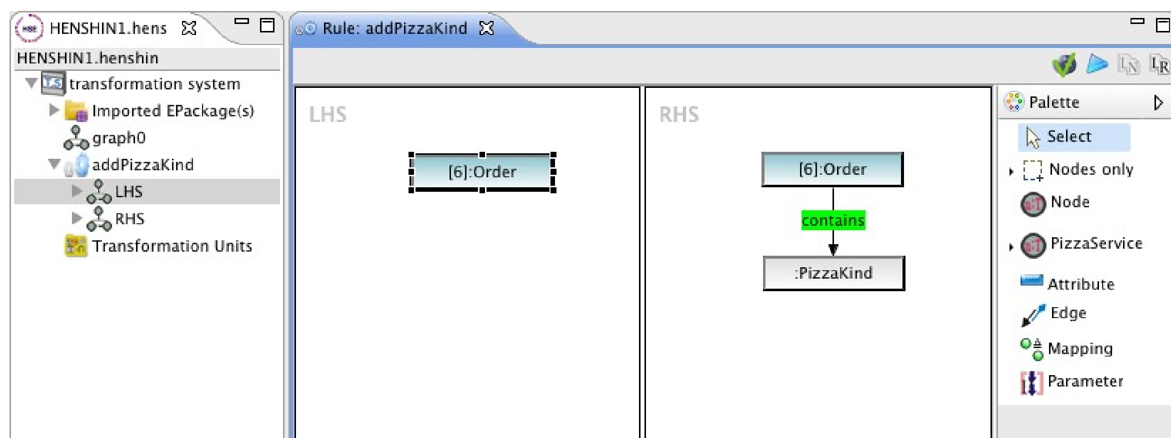


Abbildung 28

5.2 Parameter erstellen

Ein Parameter kann einen Knoten oder einen Attributwert enthalten. Wenn ein Parameter einen Knoten enthält, muss er den gleichen Namen haben, wie der Knoten. Solche Parameter können über den Menüpunkt *Create Parameter* im Kontextmenü eines Knotens in der Baum- oder Regelansicht oder über die Schaltfläche *Parameter* aus der Regelpalette erstellt werden.

Achtung: Die Option *Create Parameter* erscheint nur im Kontextmenü eines Knotens in der Regelansicht, nicht in der Graphansicht.

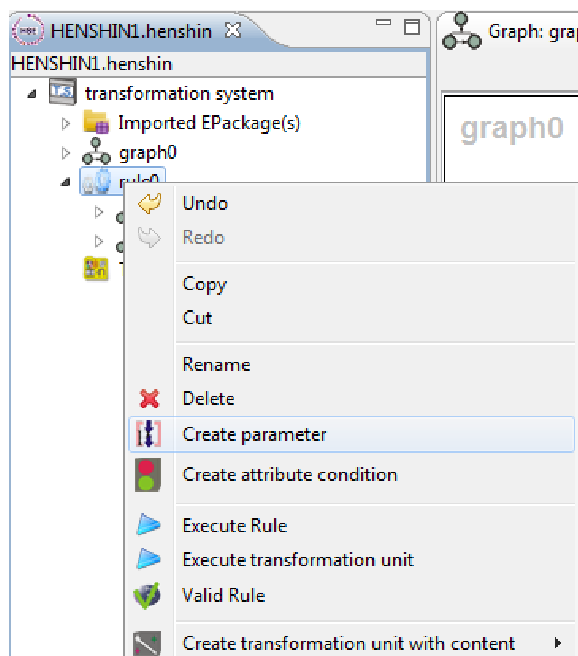


Abbildung 29: Regel-Kontextmenü

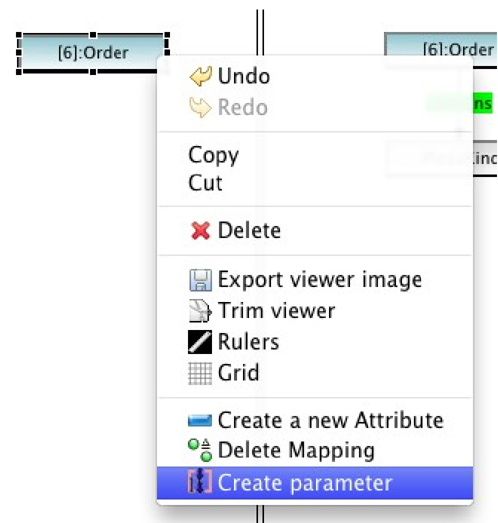


Abbildung 30: Knoten-Kontextmenü 3

Hat der ausgewählte Knoten keinen Namen, wird er von dem Benutzer abgefragt und für den Knoten und seinen Parameter gesetzt.

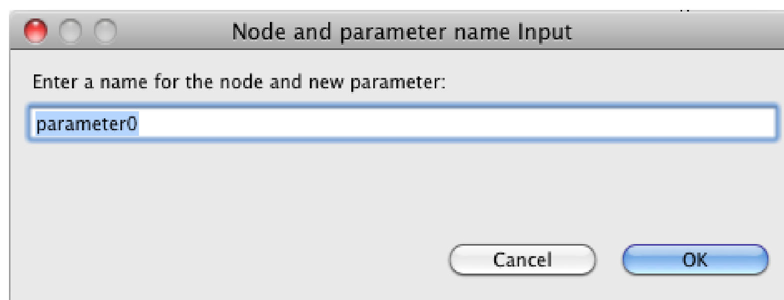


Abbildung 31: Parametername eingeben

Alternativ können Parameter auch über das Kontextmenü der Regel in der Baumansicht erstellt werden. Dabei wird der Name des neuen Parameters abgefragt (Abbildung 31: Parametername eingeben).

Beispiel: Wir erstellen zwei Parameter *s* und *p* über das Kontextmenü.

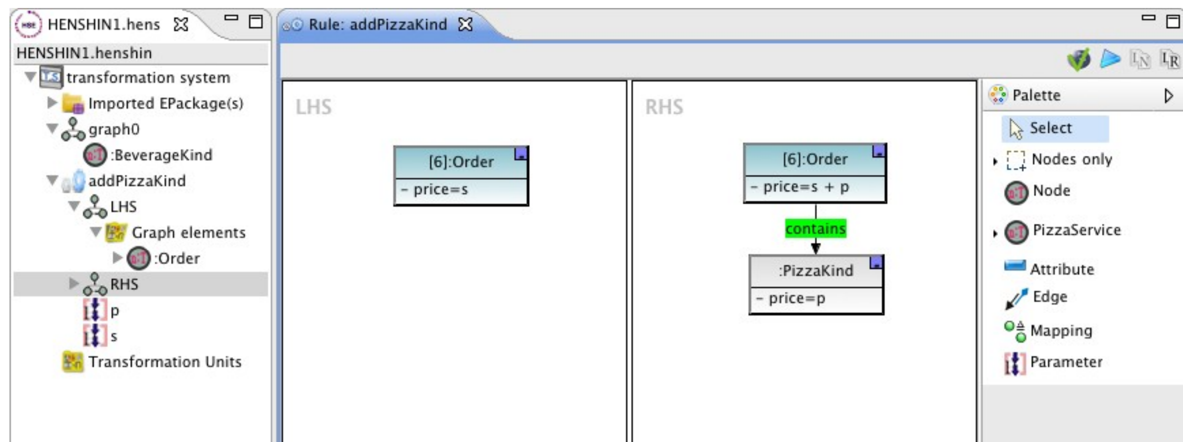


Abbildung 32

5.3 Verwendung von Attributparametern

Parameter können als Attributwerte verwendet werden. In *LHS* kann ein Parameter nur für ein Attribut benutzt werden. Dabei kann der Parameterwert vom Benutzer angegeben oder bei der Regelanwendung bestimmt werden. In *RHS* kann ein Attributwert allgemeine Java-Ausdrücke besitzen.

Beispiel: In *LHS* hat der Knoten *Order* ein Attribut *price* mit dem Wert *s*. Dieser wird bei der Regelanwendung auf den aktuellen Wert des Attributs gesetzt. Der Knoten *PizzaKind* hat ein Attribut *price* mit dem Wert *p*. Der Wert von *p* wird vom Benutzer angegeben und als Preis für das neu erstellte *PizzaKind* gesetzt. Der Knoten *Order* in *RHS* hat ein Attribut *price* mit dem Wert *s+p*. Bei der Regelanwendung wird der aktuelle Preis von *Order* um von dem Benutzer angegebenen Wert des Parameters *p* erhöht.

5.4 Attributbedingung erstellen

Eine Attributbedingung kann über den Menüpunkt *Create attribute condition* im Kontextmenü der Regel in der Baumansicht erstellt werden (Abbildung 29: Regel-Kontextmenü). Dabei werden der Name und die Bedingung abgefragt und anschließend eine neue Attributbedingung erzeugt.

Beispiel: Erzeugen einer Attributbedingung $s < 10$. Somit kann die Regel nur dann ausgeführt werden, wenn der Wert des Parameters *s* kleiner 10 ist.

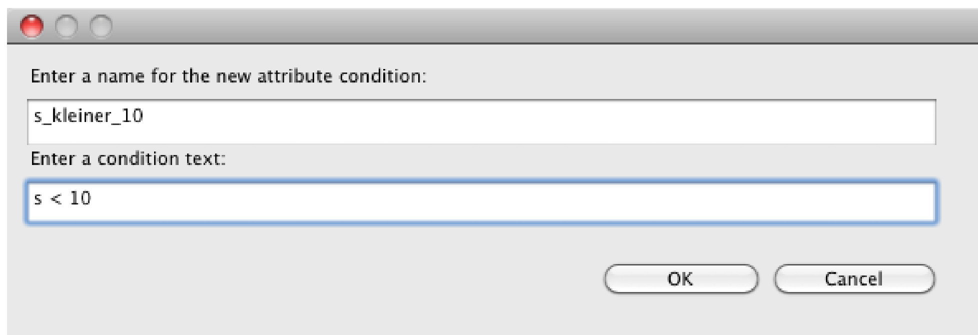


Abbildung 33: Attributbedingung erzeugen

Um den Namen und die Bedingung zu ändern, benutzen Sie die *Properties*-Ansicht. Die Bedingung kann auch in der Baumansicht entweder mit einem Doppelklick oder mit einem rechten Mausklick über den Menüpunkt *Rename* geändert werden.

6 Anwendungsbedingung einer Regel erstellen

Für eine existierende Regel kann eine Anwendungsbedingung (*Application Condition*) erstellt werden. Dabei ist es egal, ob die Bedingung über das Kontextmenü einer Regel oder über das Kontextmenü von *LHS* erzeugt wird. Hat die Regel noch keine Anwendungsbedingung, dann wird diese an *LHS* angehängt. Hat die Regel schon eine, dann wird die *Application Condition* mit der existierenden Bedingung automatisch verundet. Das Erzeugen einer Bedingung über das Kontextmenü einer *Application Condition* bedeutet, dass diese an die ausgewählte *Application Condition* angehängt wird.

Die Anwendungsbedingung von *LHS* wird in der Regelansicht ganz links als Text und als Figur dargestellt.

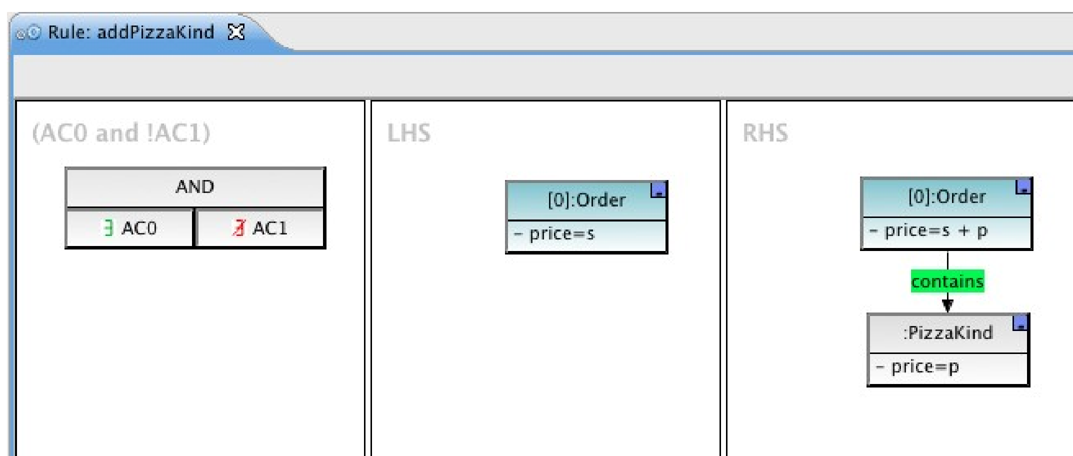


Abbildung 34: Regel-Ansicht mit Bedingung

Mit einem Doppelklick auf eine *Application Condition* in der Baumansicht bzw. in der Figur einer Regelansicht öffnet sich eine Bedingunganansicht. Auf ihrer linken Seite wird die Prämisse und auf ihrer rechten Seite die Konklusion dargestellt.

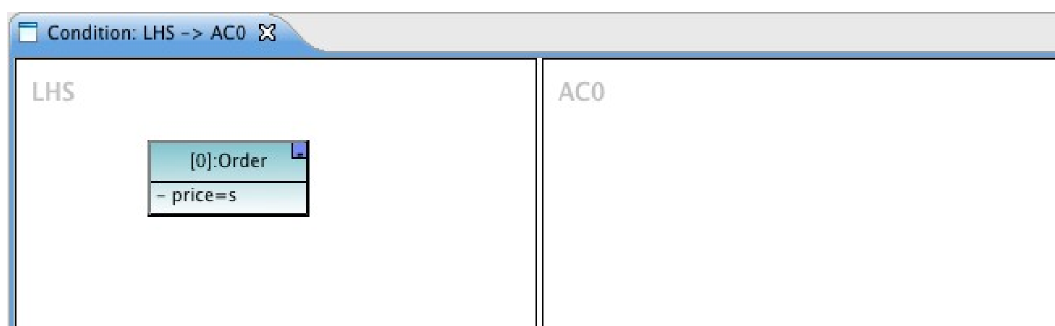


Abbildung 35: Bedingung-Ansicht

Folgende Menüpunkte dienen der Erzeugung von Anwendungsbedingungen:

- *Create Condition Tree...*
- *Create Application-Condition*
- *Create Not-Condition*
- *Create And-Condition*
- *Create Or-Condition*

Diese Menüpunkte sind nur sichtbar, wenn es erlaubt ist, eine Bedingung in das ausgewählte Objekt einzufügen.

6.1 Bedingung als Ganzes erstellen

Eine Anwendungsbedingung kann als Ganzes über *Create Condition Tree...* im Kontextmenü erstellt werden. Es erscheint ein Dialogfenster zum Zusammenbauen der Anwendungsbedingung.

Auf der linken Seite des Dialogs ist die Bedingung als Baum dargestellt. Auf der rechten Seite kann eine *Application Condition* oder eine *NOT-*, *AND-* oder *OR-Formula* in den Baum eingefügt werden.

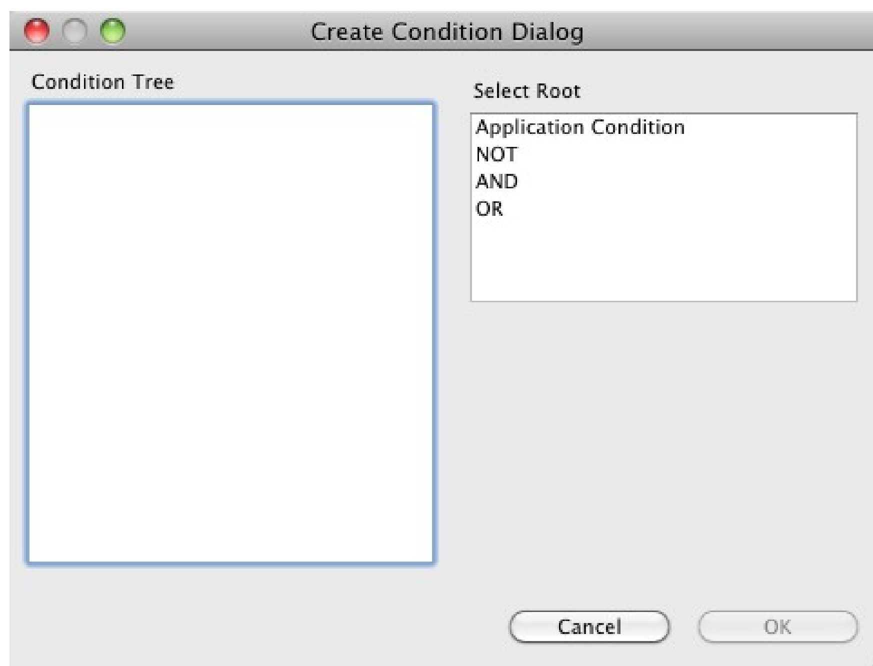


Abbildung 36: Anwendungsbedingung-Dialog

Erklärung zum Bedienen des Dialogs:

- Am Anfang ist der *Condition Tree* leer und der *OK*-Button deaktiviert. Der *OK*-Button wird erst dann aktiviert, wenn der *Condition Tree* vollständig ist, d.h. entweder ist eine *Application Condition* als Wurzel ausgewählt oder alle innersten Knoten sind *Application Conditions*. Auf der rechten Seite kann die Wurzel ausgewählt werden.

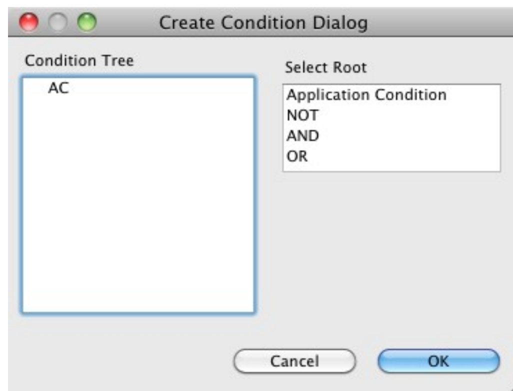


Abbildung 37: ***Application Condition*** **als Wurzel**

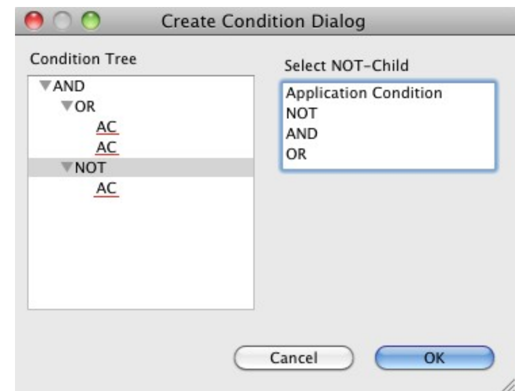


Abbildung 38: ***Application Condition*** **als innere Knoten**

- Mit *F2* können Sie eine *Application Condition* im *Condition Tree* umbenennen (siehe Abbildung 39). Um die Umbennenung zu betätigen, klicken Sie *Enter*.
- Ist eine *Application Condition* im Baum ausgewählt, erscheint auf der rechten Seite ein markiertes *negated*-Feld, was einer *NAC (Negative Application Condition)* entspricht (siehe Abbildung 39). Ist das *negated*-Feld nicht markiert, entspricht dies einem *PAC (Positive Application Condition)* .

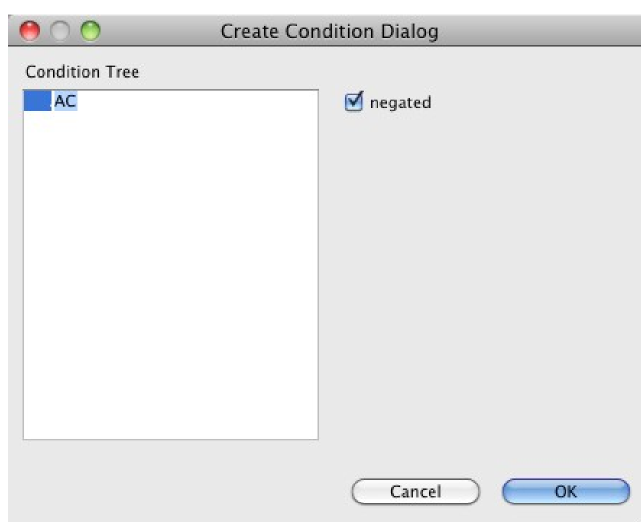


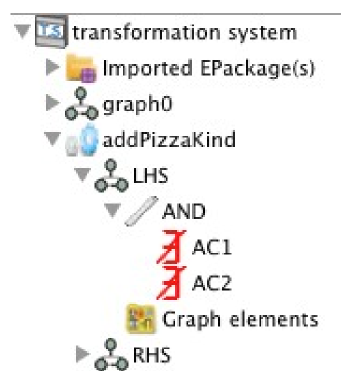
Abbildung 39

- Um einen Kindknoten an eine *NOT*-, *AND* oder *OR*-Formula anzuhängen, wählen Sie den Elternknoten auf der linken Seite. Auf der rechten Seite befindet sich eine Liste für *NOT-Child* bzw. zwei Listen für *AND-* oder *OR-Children*. Von jeder Liste wählen Sie ein Objekt, das Sie als Kindknoten von einer *NOT*-, *AND* oder *OR*-Formula festlegen.
- Wenn Sie den *Condition Tree* erneut von der Wurzel definieren wollen, entfernen Sie die Auswahl der linken Seite. Auf der rechten Seite können Sie dann die Wurzel erneut auswählen.

Beispiel: Erstellen einer Anwendungsbedingung *AND (AC1, AC2)*.

Gehen Sie wie folgt vor, nachdem das Dialogfenster angezeigt wird:

1. Wählen Sie den Menüpunkt *Create Condition Tree...* entweder im Kontextmenü einer Regel oder eines *LHS*-Graphen aus.
2. Wählen Sie *AND* auf der rechten Seite als Wurzel aus. Dieses *AND* wird auf der linken Seite als Wurzel angezeigt.
3. Wählen Sie *AND* auf der linken Seite und *Application Condition* von den beiden Listen auf der rechten Seite. Zwei *AC* sind nun an *AND* im *Condition Tree* als Kindknoten angehängt.
4. Wählen Sie die erste *AC* (bzw. die zweite *AC*), drücken Sie *F2* und benennen Sie sie zu *AC1* (bzw. *AC2*) um.
5. Nach Betätigung von *OK* schließt sich das Dialogfenster. In der Baumansicht hängt der neue Bedingungsbaum am *LHS* der entsprechenden Regel.



6.2 Bedingungen schrittweise erstellen

Um eine Anwendungsbedingung schrittweise zu erstellen, wählen Sie den gewünschten Bedingungstyp über den entsprechenden Menupünkt im Kontextmenü aus. Der neue Bedingungstyp (*AC* / *NOT* / *AND* / *OR*) ist in der Baumansicht an das ausgewählte Objekt angehängt. Unvollständige Bedingungen werden rot dargestellt. Dazu gehören *NOT-Formulas* ohne Kindknoten und *AND*- und *OR-Formulas* mit weniger als zwei Kindknoten.

Beispiel: Erstellen einer Anwendungsbedingung *AND (AC0, AC1)*.

addPizzaKind
LHS
AND

Wählen Sie eine Regel oder ein *LHS* in der Baumansicht aus und wählen Sie den Menupünkt *Create AND-Condition* im Kontextmenü eines *LHS*-Graphen aus.

addPizzaKind
LHS
AND
AC0
AC1

Wählen Sie zweimal den Menüpunkt *Create Application Condition* im Kontextmenü von *AND* aus. Die Umbenennung der *Application Condition* erfolgt entweder in der Baum-/*Properties*-Ansicht oder in der Condition-Figure der Regelansicht.

(!AC0 and !AC1)

AND	
AC0	AC1

6.3 Application Condition negieren

Der Negationswert von einer existierenden *Application Condition* kann im Kontextmenü einer *Application Condition* geändert werden. Eine *Negative Application Condition* (∄) kann über den Menüpunkt *Set negated = false* zu *Positive Application Condition* (∃) geändert werden. Umgekehrt erfolgt dies über den Menüpunkt *Set negated = true*. Der Negationswert kann außerdem in der *Properties*-Ansicht geändert werden.

6.4 Austauschen von AND und OR

Eine *AND-Formula* lässt sich mit einer *OR-Formula* austauschen. Wählen Sie den Menüpunkt *Swap AND → OR*, um eine *AND-* mit einer *OR-Formula* und den Menüpunkt *Swap OR → AND*, um eine *OR-* mit einer *AND-Formula* zu tauschen.

6.5 Formula-Mapping erstellen

Soll ein Knoten in einer Prämisse und deren Konklusion denselben Knoten im Graphen darstellen, muss für diesen Knoten ein Mapping erstellt werden.

Sie erstellen ein Mapping, indem Sie den Punkt *Mapping* aus der Palette auswählen und dann zuerst auf den Knoten in der Prämisse und danach auf den Knoten in der Konklusi-

on klicken. Gemappte Knoten erhalten dieselbe Farbe und Nummer und sind dadurch leicht als solche erkennbar.

Beispiel: Erstellen Sie ein Mapping zwischen den Knoten *Order* in *LHS* und *Order* in *AC0*.

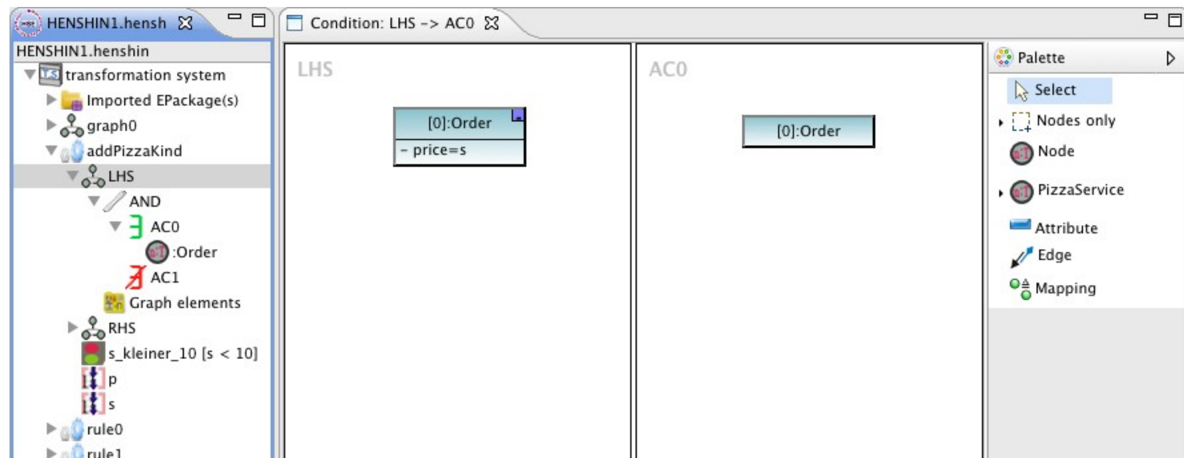
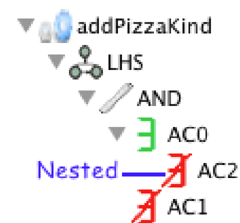


Abbildung 40

6.6 Verschachtelte Anwendungsbedingungen erstellen

Anwendungsbedingungen müssen nicht immer direkt über den *LHS*-Graphen einer Regel definiert werden, sondern können auch hierarchisch verschachtelt werden (Nested Application Conditions). Dazu wählen Sie in der Baumansicht eine *Application Condition (AC)* aus und definieren Sie dafür eine Anwendungsbedingung, wie oben beschrieben (6.1 - 6.5).

Beispiel: Definieren Sie eine Anwendungsbedingung *AC2* über die Bedingung *AC0* (siehe Abbildung 40). Wählen Sie *AC0* in der Baumansicht aus und wählen Sie den Menupünkt *Create Application Condition* in ihrem Kontextmenü aus.



7 Transformation-Units erstellen

Eine leere Transformation-Unit kann in der Baumansicht über das Kontextmenü des Transformationssystems oder des Ordners *Transformation Units* erstellt werden.

Wählen Sie den Menüpunkt *Create transformation unit with content* und danach den gewünschten Typ der Unit von den entsprechenden Menüpunkten aus.

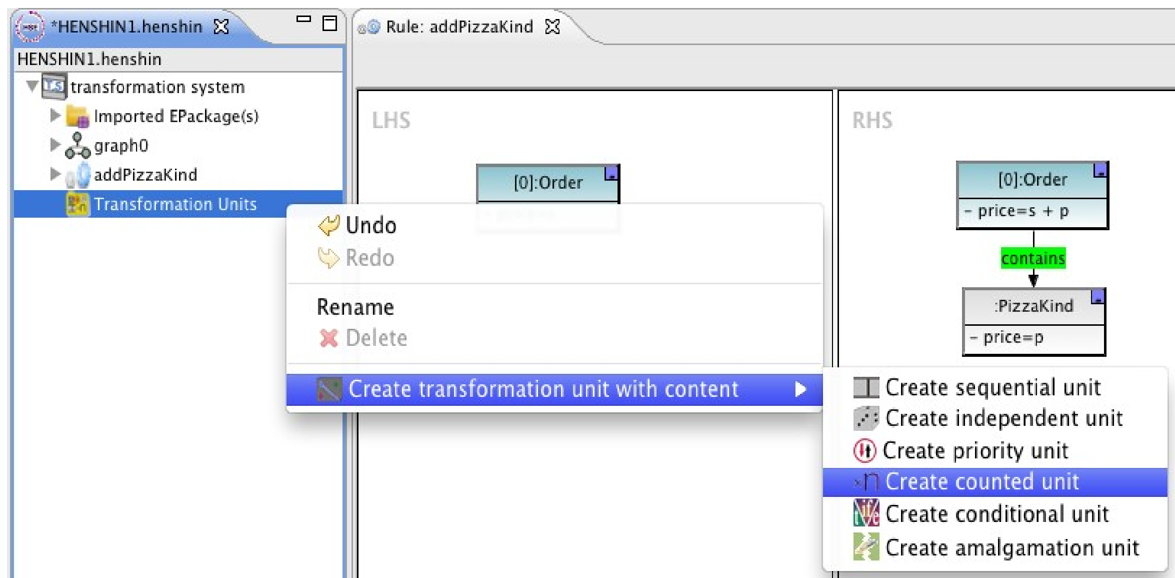


Abbildung 41: Transformation Unit-Kontextmenü

Der Name der zu erstellenden Unit wird in einem Dialogfenster abgefragt.

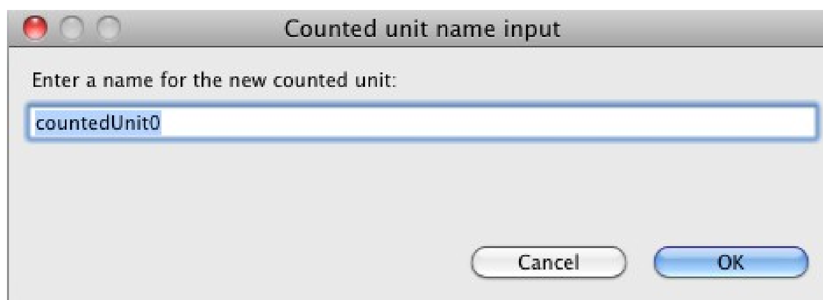


Abbildung 42: Transformation Unit Name eingeben

Die Namen aller Transformation-Units eines Transformationssystems müssen eindeutig sein. Der Henshin-Editor stellt dies sicher, indem bereits vorhandene Namen zurückgewiesen werden. Nach Bestätigung des Dialogs wird die Transformation-Unit-Ansicht der neuen Unit geöffnet.

Beispiel: Erstellen Sie eine neue Counted-Unit. Wählen Sie hierfür den Menüpunkt *Create counted unit* aus. Die Transformation-Unit-Ansicht wird automatisch geöffnet und zeigt eine leere Counted-Unit-Figur.

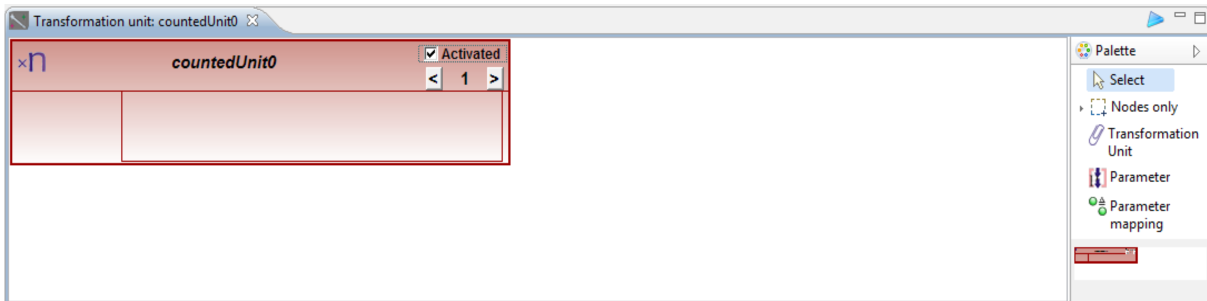


Abbildung 43: Transformation Unit-Ansicht

7.1 Transformation-Unit hinzufügen

Eine Transformation-Unit kann über den Menüpunkt *Create Transformation Unit* im Kontextmenü von einer beliebigen anderen Transformation-Unit oder über die Schaltfläche *Transformation Unit* aus der Palette eingefügt werden.

Wenn eine Container-Unit bereits andere Units enthält, kann beim Hinzufügen über die Palette die Position unmittelbar bestimmt werden. Das bedeutet, wenn eine neue Unit am Ende einer Container-Unit hinzugefügt werden soll, müssen Sie nur auf die Schaltfläche *Transformation Unit* klicken und anschließend auf die Parent-Unit selbst. Wenn Sie eine bestimmte Position haben wollen, klicken Sie auf die untergeordnete Unit, die die gewünschte Position im Container besetzt. Falls mehrere hinzufügende Units existieren, wird eine Auswahl angeboten.



Abbildung 44: Unit Auswahl

Beispiel: Einfügen einer Transformationsregel *rule0* in der zuvor erstellten Counted Unit über das Kontextmenü *Add Transformation Unit*.

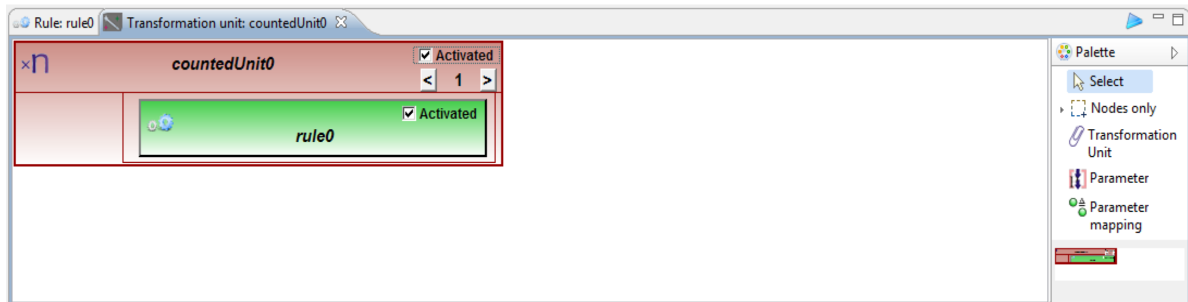


Abbildung 45: Transformation-Unit einfügen

7.2 Parameter erstellen

Einen Parameter können Sie über den Menüpunkt *Create Parameter* im Kontextmenü oder über die Schaltfläche *Parameter* aus der Palette erstellen. Wenn Sie den Parameter über die Palette erstellen, müssen Sie danach auf die Transformation-Unit klicken, für die er erstellt werden soll. Bei der Erstellung eines neuen Parameters wird der Name über einen 3,26Dialog abgefragt (siehe Abbildung 31, S. 121).

Beispiel: Erstellen eines neuen Parameters *price*.

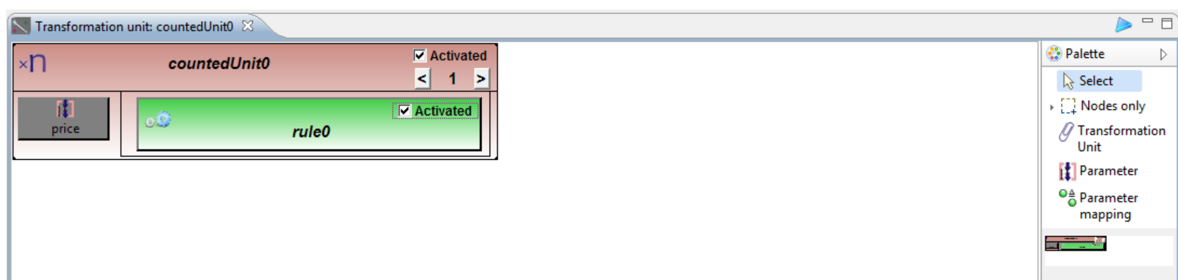


Abbildung 46: Erstellen eines Unit-Parameters

7.3 Sub-Unit öffnen

In der Transformation-Unit-Ansicht können gleichzeitig eine Parent-Unit und eine ihrer Sub-Units angezeigt werden. Eine Sub-Unit kann geöffnet werden, indem sie doppelt angeklickt wird. Dabei wird die Transformation Unit, deren Sub-Unit geöffnet werden soll, links angezeigt und die Sub-Unit rechts. Auf diese Weise kann eine Unit beliebig tief geöffnet werden. Mit einem Doppelklick auf das oberste Element der links angezeigten Unit gelangen Sie auf die obere Unit zurück.

Beispiel: Öffnen Sie die Ansicht der Sub-Unit *rule0* , indem Sie sie in der Figur doppelt anklicken.

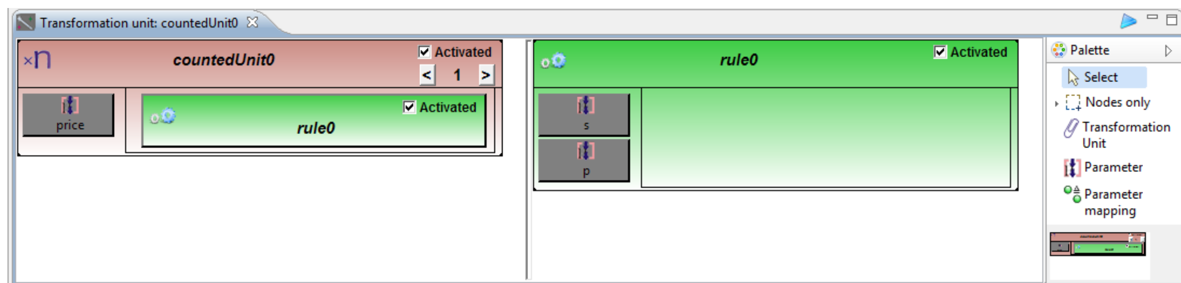


Abbildung 47: Öffnen der Sub-Units

7.4 Mapping zwischen Parametern erstellen

Wenn eine Transformation-Unit und ihre Sub-Unit geöffnet sind, kann zwischen ihren Parametern ein Mapping erstellt werden. Gehen Sie hierzu wie folgt vor:

1. Wählen Sie den Menüpunkt *Parameter mapping* aus der Palette.
2. Klicken Sie auf den Parameter, von dem der Wert übernommen werden soll.
3. Klicken Sie auf den Parameter, an den der Wert übergeben werden soll.

Das Mapping wird in der Parent-Unit als Pfeil von ihrem Parameter zu ihrer Sub-Unit dargestellt, die den zweiten Parameter der Mappings enthält. Die Richtung des Pfeils zeigt die Richtung der Wertübergabe an. Gemappte Parameter erhalten dieselbe Farbe und sind dadurch leicht als solche erkennbar.

Beispiel: Erstellen eines Mappings vom Parameter *price* zum Parameter *p*.

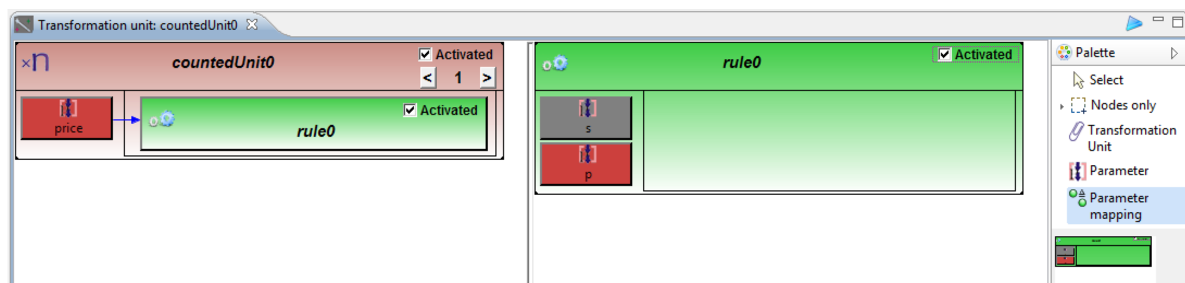


Abbildung 48: Parameter Mapping

7.5 Reihenfolge der Sub-Units ändern

Wenn eine Transformation-Unit mehrere Sub-Units enthält, kann ihre Reihenfolge per „Drag and Drop“ geändert werden. Klicken Sie die Sub-Unit mit der Maus an und halten Sie die Maustaste gedrückt. Ziehen Sie nun die Sub-Unit auf die gewünschten Position und lassen Sie die Maustaste los.

7.6 Transformation Unit mit Inhalt erstellen

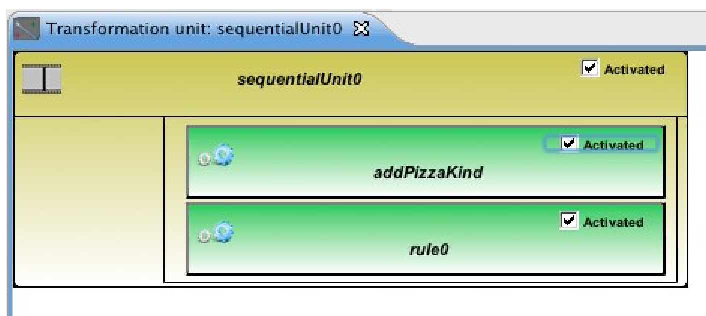
Um eine Transformation-Unit mit Inhalt zu erstellen, gehen Sie wie folgt vor:

1. Markieren Sie ein oder mehrere Transformation-Units.
2. Klicken Sie die markierten Transformation-Units mit der rechten Maustaste an. Es öffnet sich ein Kontextmenü.
3. Wählen Sie im Kontextmenü den Menüpunkt *Create transformation unit with content*. Es öffnet sich ein Untermenü mit allen Transformation-Unit-Typen, die die aktuell markierten Units enthalten können.
4. Wählen Sie eine der angebotenen Unit aus. Es wird eine neue Transformation-Unit erzeugt, die die markierten Units enthält.

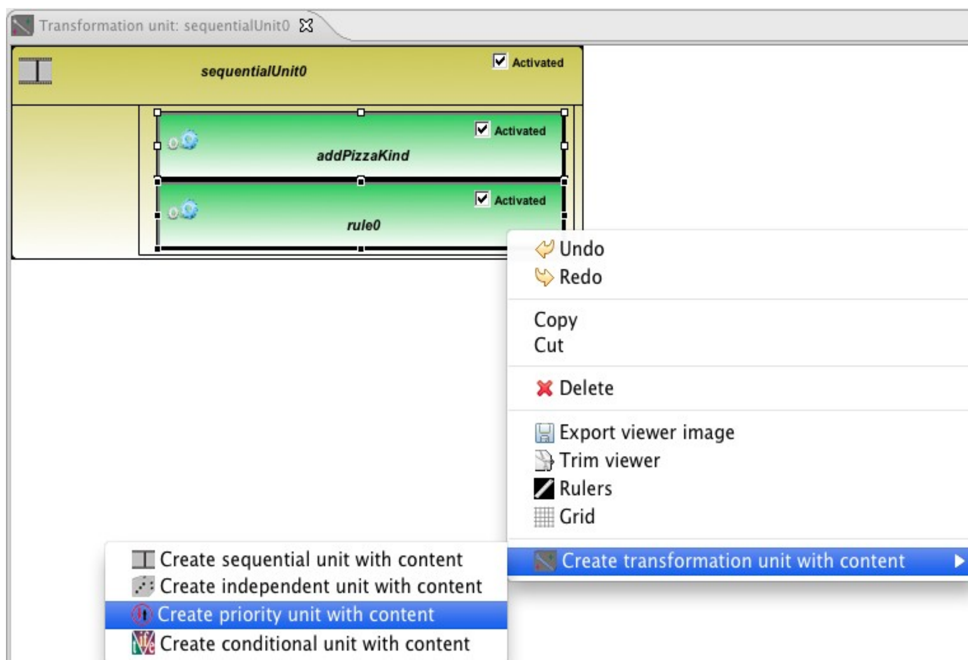
Achtung: Wenn Sie diese Aktion aus einer anderen Transformation-Unit ausführen, werden die markierten Units mit der neu erstellten Unit ersetzt. Bei der Ersetzung werden alle existierende Mappings der Parameter über die neu erstellten Parameter weitergeleitet.

Beispiel:

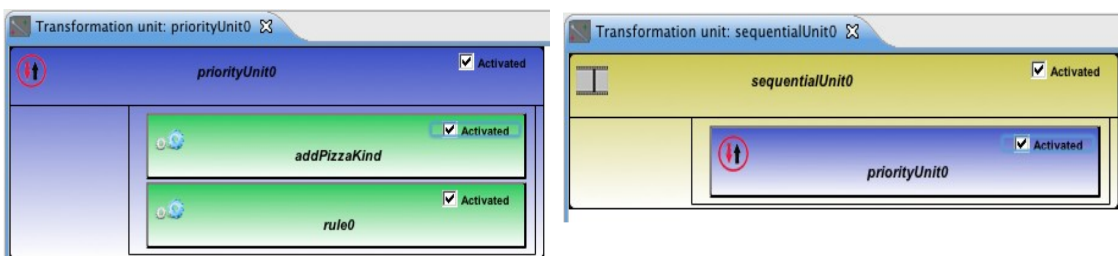
1. Ausgangsposition: *sequentialUnit0* enthält die zwei Regeln *addPizzaKind* und *rule0*.



2. Markieren Sie die beiden Regeln.
3. Öffnen Sie mit der rechten Maustaste das Kontextmenü und wählen Sie dort *Create priority unit with content* aus.



4. Eine Priority-Unit mit dem Inhalt *addPizzaKind* und *rule0* wird erstellt und die *sequentialUnit0* enthält jetzt diese Priority-Unit.



7.7 Ausführungsanzahl des Counted Units ändern

In einer Counted Unit können Sie in dem obersten Teil der Unit-Figure die Ausführungsanzahl bestimmen. Dazu dienen die Pfeil-nach-links- und Pfeil-nach-rechts-Tasten. Bei der Erzeugung wird der Counter automatisch auf 1 gesetzt. Ein unendliches Ausführen einer Counted Unit kann erreicht werden, indem Sie die Ausführungsanzahl auf *-1* setzen.

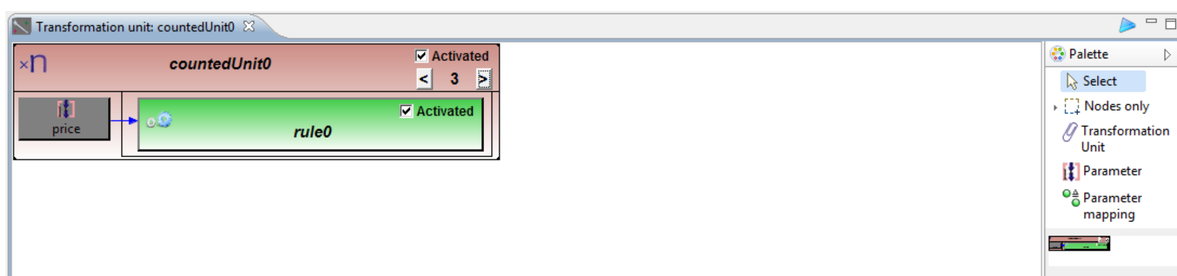


Abbildung 49: Ausführungszahl ändern

Bespiel: Erhöhen der Ausführungsanzahl einer Counted-Unit auf 3.

7.8 Amalgamation-Unit erstellen

Eine Amalgamation-Unit ist ein Spezialfall einer Transformation-Unit. Diese Unit besteht aus genau einer Kernregel und mehrerer Multiregeln.

Wie alle Units, können Amalgamation-Units über den Menüpunkt *Create transformation unit with content* und den Unter-Menüpunkt *Create amalgamation unit* im Kontextmenü eines Transformationssystems oder des Containers *Transformation Units* erstellt werden (Abbildung 41, S. 131).

Eine Besonderheit bei der Erstellung dieser Unit ist, dass Sie zuerst entweder eine neue Regel über den Menüpunkt *Create Kernel Rule* oder eine bestehende Regel über *Add Defined Rule as Kernel* im Kontextmenü einer Amalgamation-Unit als Kernregel definieren müssen, bevor Sie eine Multiregel über *Create Multi Rule* erstellen können.

Sobald eine Amalgamation-Unit eine Kernregel besitzt, verschwindet der Menüpunkt *Create Kernel Rule*. Verfügt eine Amalgamation-Unit bereits über eine oder mehrere Multiregeln, wird der Menüpunkt *Add Defined Rule as Kernel* nicht mehr angezeigt. Dadurch wird sichergestellt, dass eine Amalgamation-Unit nur eine Kernregel besitzen darf.

Bei der Erzeugung einer Multiregel werden alle Knoten und Kanten von der Kernregel in die Multiregel kopiert. Alle Änderungen in der Kernregel, wie das Hinzufügen neuer Knoten und Kanten, Löschen, Mappingerstellen, Parametererstellen usw., werden in der Multiregel auch ausgeführt. Jedoch werden Änderungen, die in der Multiregel vorgenommen werden, nur in dieser ausgeführt.

In der Multiregelansicht lassen sich die Knoten von Kern- und Multiregeln dadurch unterscheiden, dass die Knoten der Kernregel leicht verblassend und die der Multiregel mit zwei hintereinander versetzten Recktecken (genannt „Multiknoten“) dargestellt sind. Die von der Kernregel kopierten Knoten dürfen in der Multiregel nicht geändert werden. Die Multiknoten eines *LHS*-Graphen können in *RHS* durch Klicken des Symbols (rechts oben) in der Werkzeugleiste kopiert werden.

Beispiel: Definieren Sie eine amalgamierte Regel *amalgamationUnit0* , deren Kernregel (*rule0*) einem Kunden (*Customer*) eine Bestellung (*Order*) zuweist. Die amalgamierte Regel soll eine Multiregel (*rule1*) haben, bei der zu jeder bestellten Pizza (*PizzaKind*) ein Getränk (*BeverageKind*) mitbestellt werden soll.

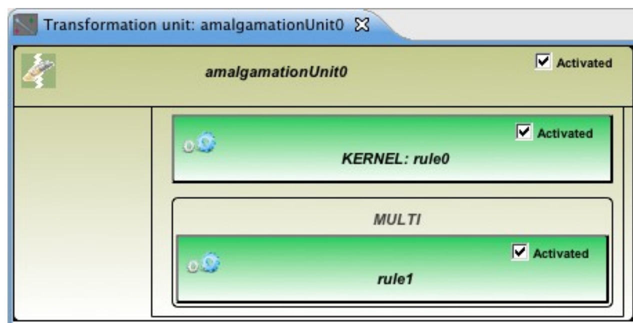


Abbildung 50: Beispiel einer Amalgamierte Regel

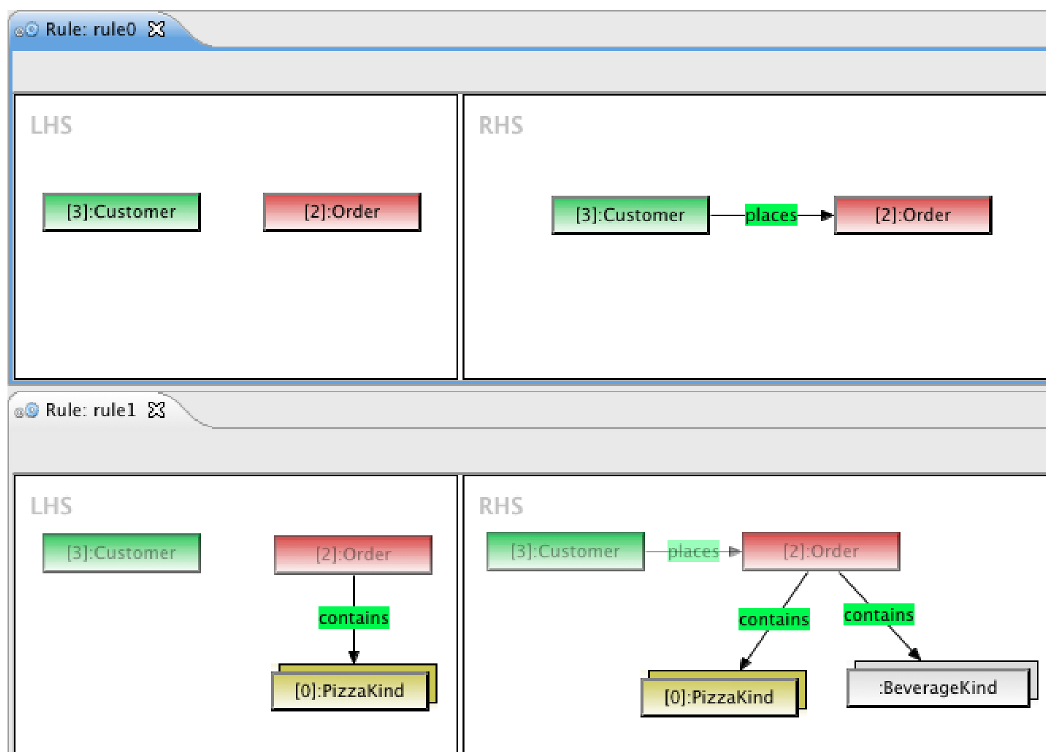


Abbildung 51: Beispiel einer Kern- und einer Multiregel

8 Elemente löschen

Alle Elemente, die in einer Baumansicht oder graphischen Ansicht als ein Objekt dargestellt sind und ausgewählt werden können, lassen sich mit Hilfe der *Delete*-Schaltfläche in der Wertzeugsleiste oder des Menüpunktes *Delete* löschen.

Eine Ausnahme bilden die Mappings zwischen den Knoten in einer Regel, da diese nur durch eine Einfärbung (bzw. Nummerierung) gekennzeichnet sind. Die Mappings der Knoten können nur über das Kontextmenü des Knotens *Delete Mapping* (Abbildung 30, S. 112) in der Regelansicht entfernt werden.

Achtung: Das Löschen von Parametermappings erfolgt nicht durch das Auswählen von *Delete Mapping*, wie bei Knotenmapping. Um die Parametermappings zu löschen, klicken Sie auf den Mappingpfeil in der Transformation-Unit-Figure und wählen Sie die *Delete*-Schaltfläche in der Werkzeugsleiste oder den Menüpunkt *Delete*.

9 Transformation Regel bzw. Unit ausführen

Die Ausführung der Transformationsregel kann über

- den Menüpunkt *Execute Rule* oder *Execute transformation unit* im Kontextmenü eines Graphen oder einer Regel (siehe 1 in Abbildung 52) oder
- aus der Werkzeugleiste (siehe 2 und 3 in Abbildung 52) einer Transformationsregel, einer Tranformation-Unit oder eines Graphen, auf denen die Transformation durchgeführt wird

gestartet werden.

Es gibt zwei Alternativen, wie Sie eine Transformationsregel oder eine Transformation-Unit starten können:

- Aus einer Transformationsregel oder einer Transformation-Unit

  Wenn im Transformationssystem nur ein Graph existiert, wird die Regel bzw. Unit auf diesen angewendet. Ansonsten werden die existierenden Graphen im Dialogfenster angezeigt. Wählen Sie dort den Graphen aus, auf den die Regel oder Unit angewendet werden soll.

- Aus einem Graphen

  Wenn im Transformationssystem nur eine Regel bzw. eine Unit existiert, wird die Regel bzw. Unit auf den selektierten Graphen angewendet. Ansonsten werden die existierenden Regeln und Units im Dialogfenster angezeigt. Wählen Sie dort die Regel oder die Unit aus, die auf den Graphen angewendet werden soll.

Wenn die auszuführende Transformationsregel bzw. Transformation-Unit Parameter enthält, versucht der Henshin-Editor zuerst die Parameterbelegung selbstständig zu ermitteln. Findet er keine, öffnet sich ein Dialogfenster zum Belegen der Parametern.

Nach der Ausführung einer Transformation-Unit werden alle angewendeten Transformationsregeln als Historie (siehe 4 in Abbildung 52) in dem Graphen angezeigt. Um die Regelanwendung zurückzusetzen und dann wieder anzuwenden, wählen Sie eine Regel aus der Historie aus und klicken Sie sie doppelt an. Dabei wird diese Regel und all ihre Vorgänger angewendet. Die nachfolgenden Regeln werden zurückgesetzt. Das Zurücksetzen aller Regelanwendungen erfolgt mit einem Doppelklick auf den Namen der Transformation-Unit in der Historie.

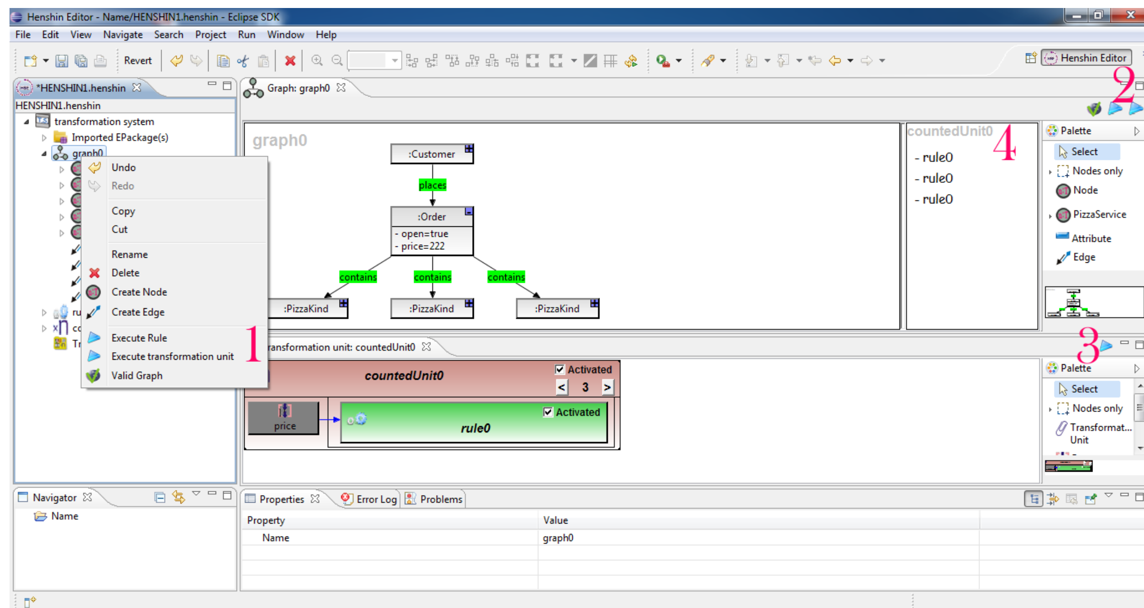


Abbildung 52: Transformationsregel bzw. Unit ausführen

10 Prüfen der Gültigkeit von Regeln bzw. Graphen

Ein Graph kann auf die EMF-Kompabilität überprüft werden. Dies kann über den Menüpunkt *Valid Graph* im Kontextmenü eines Graphen (siehe 1 in Abbildung 53) oder aus der Werkzeugleiste (siehe 2 in Abbildung 53) eines Graphen gestartet werden.

Die Überprüfung auf EMF-Kompabilität einer Regel kann entsprechend über den Menüpunkt *Valid Rule* im Kontextmenü einer Regel oder aus der Werkzeugleiste einer Transformationsregel (siehe 3 in Abbildung 53) gestartet werden.

Nach der Überprüfung werden die Validierungsergebnisse in einem Dialog angezeigt.

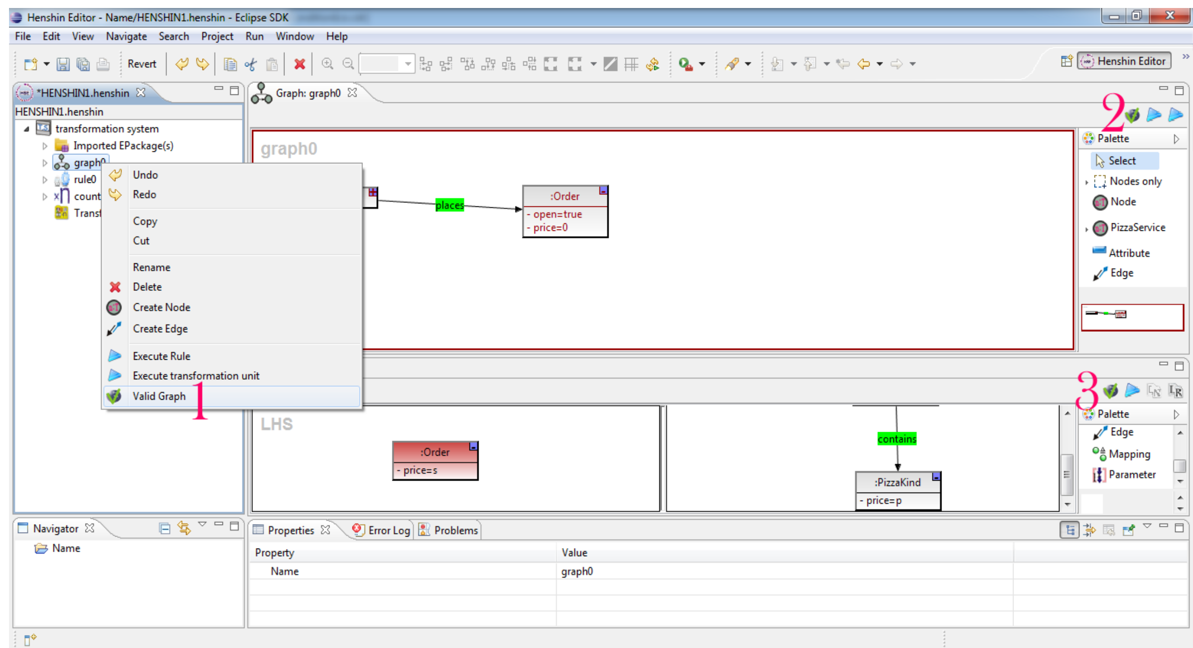


Abbildung 53: Transformationsregel bzw. Graph validieren